Concurrent object-oriented execution of OPS5 production systems

Odeh, Mohammed Hosni

Award date:
1993

Awarding institution:
University of Bath

Concurrent Object-Oriented Execution of OPS5 Production Systems

submitted by

Mohammed Hosni Odeh

for the degree of Ph.D.

of the

University of Bath

1993

Signature of Author .....

Mohammed Hosni Odeh

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

Over the last decade, Artificial Intelligence (AI) has been the aim of many researchers in the field of expert systems and particurally the issue of knowledge representation. The OPS5 production system represents a model that is widely used in real applications of expert systems. Unfortunately, most production systems suffer one inherent problem in that they are not fast enough to be used in critical environments where quick response time is an issue. The primary goal of this thesis is to improve the performance of OPS5 production systems through concurrent execution. The thesis reports the experience of taking a well-known and quite a complex problem—the OPS5 inference engine—and reconstructing it with objects in significant depth by means of Booch's object-oriented development methodology. Unlike previous research attempts at improving the performance of OPS5 production system programs, the use of object-oriented technology resulted in constructing a software architecture that is parallelisable, extensible and does not rely on special-purpose hardware. This has led to synthesize a new object-oriented inference engine which has been named, OOPS5.

OOPS5 employs a new algorithm to execute OPS5 production system programs which utilises an important principle in that it avoids the recomputation of join between condition elements of the same production. A significant corollary from the use of objects is that productions may be added incrementally and matched by the existing configuration of WM elements whereas the Rete network requires the recompilation of the give production system program. The performance of the serial OOPS5 is comparable with that of Rete for large and real applications of expert systems.

The concurrent OOPS5 uses the futures model of concurrency and shows that OOPS5 is suitable for parallel implementations. Moreover, the empirical results of the concurrent OOPS5 seem to be in line with published simulations of intra-node and action level parallelism rather than the simulation speed-ups of special architectures. This makes OOPS5 a real, rather than simulated, contribution to the parallel execution of OPS5 programs. In conclusion, OOPS5 is a concurrent object-oriented platform to experiment with various ideas in concurrent processing which may contribute further to improving the performance of production systems.

Acknowledgements

I would like to thank my supervisor, Dr. Julian Padget for his guidance, encouragement and support during the various stages of this research and in particular during my illness. Thanks are also extended to Professor John Fitch for his valuable advises and support especially in the initial stage of this research. I would also like to thank the past and present members of the computing group at the University of Bath and especially Dave Hutchinson, Icarus Sparry and Pete Broadbery.

This research has been partially supported by Arab Petroleum Investments CORPoration (APICORP). I am very grateful to Dr. Nurredin Farrag, the general manager of APICORP, for his continuous support, sincere encouragement and continuous interest, especially during my illness. Special thanks are also extended to Mr. Galal Osman for his support and appreciation of research. I must also mention Mr. Farouq Hassan for his valuable advises and comments and Mr. Maher Elbouhi for his great help in the administrative work.

I am very grateful to Mr. Khader Herzallah and his son Hatem—a childhood friend—for their encouragement and support. I would like to thank Mr. Mohammed Khulaif, my English language teacher, for his teaching of the language and his continuing interest since then. I would also like to thank all my friends in Bath and especially Dr. Mohammed Dulaime and his family for their support of my family and making them feel home in the difficult circumstances.

I am grateful to my brothers Hani and Yasser whose support, care and criticisms are never-ending and particularly Hani. And, not to forget my sisters for their prayers and good wishes for me and my family. I also thank my father in-law for his care and attention and my brothers in-law for their valuable comments especially Mohammed Hajj.

Last, but not least, my sincere gratitude to my wife who sacrificed her time and helped in bringing this research to a safe conclusion. To her and my children Omar, Abdul-Rahman and Hamzah is my deep appreciation for their patience and sacrifice.

To **my beloved parents**

who spent their lives until their death educating me, my sisters and brothers.

May all the mercy of *Allah* be upon them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decade, Artificial Intelligence (AI) has been the aim of many researchers and particularly in the field of expert systems or knowledge-based systems. Among the main issues that have been investigated in expert systems research is knowledge representation. In particular, the production system represents a model of knowledge representation that is mostly applied in real applications of expert systems such as MYCIN [41], for diagnosing bacterial infections of blood, DENDERAL [4] for deducing chemical structures from mass spectrometry data, R1 [28] as an automatic configurer of VAX computer systems and others. Davis and King [7] suggested that production systems can be used in problem domains where knowledge is diffuse, independent from the way it is to be used and processes can be represented as a set of independent actions. Moreover, Rychener [39] proposed that production systems are appropriate for problem domains where a given task can be viewed as a transition from one state to another in a problem space which may be simulated by one or more production firings.

Unfortunately, most production systems suffer one inherent problem in that they are not fast enough to be used in critical environments where quick response time is an issue. The primary goal of this research is to improve the performance of production rule systems through concurrent execution. There has been much work on this [15, 31] with an emphasis on the insertion of parallel constructs into an existing, relatively sequential, program.

In this research, a new approach has been proposed and implemented to execute

production systems based on object-oriented transformation of such systems so as to establish a platform for concurrent object-oriented execution of production systems. OPS5 [10], which is a production system building language, is used in this research for two reasons. First, many expert system applications have been written using OPS5; secondly most of the research being carried out towards improving performance of production systems model is based on OPS5.

In this research, instead of starting from the existing program, namely OPS5, we began from a specification of that program derived from an informal analysis of the semantics of OPS5 rule sets and applied a well-known object-oriented methodology to define classes and methods on them to reconstitute an OPS5 interpreter.

A significant corollary of the use of objects is that productions may be added incrementally and matched with the current configuration of WM elements, whereas OPS5's original match algorithm (i.e. Rete) cannot match these productions with the previously added working memory elements because this requires recompilation of the rule set.

Static and dynamic measurements have been done to evaluate this implementation and one of the main conclusions drawn is that production systems may have different behaviours at run-time mainly because of the different static characteristics of their productions and the way they are programmed.

Also, this research addresses the future message passing mechanism through the use of the futures construct in EuLisp [34] to implement the concurrent execution of the object-oriented implementation of the OPS5 production rule system which has been named OOPS5.

1.1 Overview Of Production Systems Architecture

In general, the architecture of the production systems model consists of three components, Production Memory (PM), Working Memory (WM) and an Inference Engine as shown in Figure 1-1. Since many of the production systems applications are written in OPS5 [10, 3], which is a production system shell, and many of the efforts towards

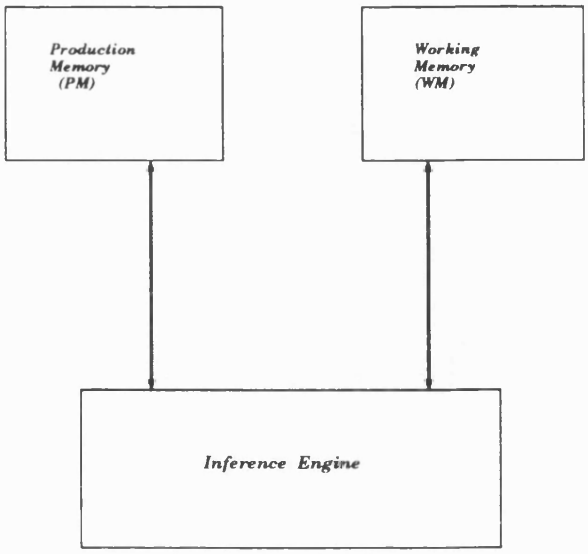

Figure 1-1: Architecture of Production Systems Model

improving production systems performance are based on this language, OPS5 will be used as a tool to present each of these components in detail.

1.1.1 Working Memory (WM)

In OPS5, Working Memory(WM) consists of a set of ordered pairs, each having the following form:

<timetag, WM element>

where *timetag* is an integer associated with each *WM element* to indicate when it was inserted into WM. A WM element is a list of elements, where the first element is a class name of a certain entity (e.g. Employee, Book) and the rest of the elements are attribute-value pairs of that class. A class is normally defined at the beginning of an OPS5 production system program using the `literalize` statement. For example, the `Monkey` entity in the Monkey and Bananas production system program [3] is defined as follows:

```
(literalize Monkey at on holds)
```

where `at`, `on` and `holds` are attributes of the `Monkey` class. Each of the attribute-value pairs in a WM element is of the following form:

^ *attribute value*

where,

^ indicates to OPS5's interpreter that the following symbol is an attribute of a class.

attribute is the name of an attribute that belongs to a certain entity defined previously using a `literalize` statement.

value should be either a symbolic or numeric constant

For example, the following is a WM element of the `Monkey` class:

```
(monkey ^at 5-7 ^on couch ^holds banana)
```

1.1.2 Production Memory (PM)

Production Memory (PM) consists of a set of productions. The representation of a production stems from the natural conditional If-Then statement. As a result, a production has two parts: antecedent and consequent, or in other terminology Left-Hand Side (LHS) and Right-Hand Side (RHS). The antecedent consists of a number of condition elements and the consequent consists of a number of actions as shown below:

C_1 & C_2 & C_3 & ... & C_n — A_1 ; A_2 ; A_3 ; ... A_m.

In OPS5, each condition element is a list of elements, where the first element is a class name of a certain entity defined as shown in the previous section and the rest of the elements are attribute-predicate-value triples. Each of these triples in a condition element of a particular production has the following form:

^*attribute predicate value*

where,

attribute is the name of an attribute that belongs to a certain entity.

predicate is one of the following:

```
(p mb17
      (goal ^status active ^type on ^object <o>)
      (object ^name <o> ^at <p>)
      (monkey ^at <p> ^holds nil)
   -->
      (write (crlf) climb onto <o>)
      (modify 3 ^on <o>)
      (modify 1 ^status satisfied))
```

Figure 1-2: Example of an OPS5 production

=	to denote numeric or symbolic equality
<>	to denote numeric or symbolic inequality
>	to denote the numeric equality "greater than"
<	to denote the numeric equality "less than"
>=	to denote the numeric equality "greater than or equal to"
<=	to denote the numeric equality "less than or equal to"
<=>	to denote not same type predicate

value can be of two types: (1) Constant as either numeric or symbolic (2) Variable which is represented by a symbolic atom preceded by a "<" character and followed by ">" a character (e.g. <x>).

In OPS5, actions in the RHS of productions are of three types: (1) WM actions such as adding, removing or modifying a WM element. (2) I/O actions (3) other actions such as function calls. Figure 1-2 presents an example of an OPS5 production, namely production mb17 of the well known Monkey and Bananas program [3].

1.1.3 Inference Engine

The inference engine is that part of the production system interpreter that executes productions. OPS5's inference engine executes productions in a cycle known as *Recognize-Act* [8]. This cycle consists of three phases: *matching*, *conflict-resolution* and *action*, where recognize refers to the first two. The following is a summary of the processing performed by each of these three phases:

(1) *Match Phase:* In this phase, the inference engine evaluates condition elements of all productions against the current contents of WM according to a predefined match algorithm. OPS5 uses the Rete match algorithm [8]. The resulting matched productions along with WM elements matching each production are collectively referred to as the conflict set or as productions instantiations.

(2) *Conflict-Resolution Phase:* The output of the match phase is the input to this phase, where one production instantiation is selected for firing (i.e. to be executed) according to a predefined conflict-resolution strategy. OPS5 offers two conflict-resolution strategies LEX and MEA which are based on recency of WM elements (i.e. with respect to time tags of WM elements) and specificity of WM elements (i.e. number of constant and variable tests in the LHS of a production). For more details, see [10]. If the conflict set is empty then execution halts.

(3) *Act Phase:* The selected production instantiation from phase 2 is fired by executing its RHS. If any of the RHS actions specify halt, then the inference engine stops. Otherwise, the inference engine executes another Recognize-Act cycle.

1.2 Organisation of the thesis

This thesis is divided into six chapters in a way to give the reader a consistent and gradual view of the work that has been carried out.

After the introduction to the thesis and the overview of the architecture of the production system model in this chapter, a survey on the efficiency of production systems is presented in chapter 2 to summarize most of the efforts made by researchers to date to improve the performance of production systems in both uniprocessor and multiprocessor environments. Research in the uniprocessor environment may be summarised as either trying to find better algorithms (e.g. Rete [8], TREAT [31]) or exploiting better architectures (e.g. RISCF [24]). On the other hand, research in multiprocessor environments may be described as exploiting parallelism in OPS5 production systems by proposing parallel architectures along with parallel algorithms to be run on them.

Chapter 3 presents a new method that has been implemented for executing pro-

duction systems based on object-oriented transformation of such systems. To help in understanding, building and presenting this transformation, an object-oriented design methodology proposed by Booch [1] has been followed step by step. This has resulted in developing an object-oriented transformation engine which accepts as an input an OPS5 production system and produces as an output an object-oriented system to be executed in an object-oriented environment. This approach has been implemented and executed using the EuLisp [34] language.

Chapter 4 presents the results of executing a set of well-known OPS5 production systems of varying characteristics to verify this new approach. Each of these systems was transformed into an object-oriented system and then two types of measurements were gathered, static and run-time. Static measurements are concerned with static information which is gathered during the transformation of OPS5 production system programs into object-oriented ones. Dynamic measurements refer to measurements that are gathered during the execution of the object-oriented transformed OPS5 production system programs. One of the main results of these measurements is a set of performance measurement tools that are valuable for understanding the bottlenecks in the performance of production systems and are easy to implement.

Chapter 5 discusses the concurrent object-oriented execution of OPS5 production system programs using the future message passing mechanism to implement the following four levels of parallelism: condition-element parallelism, join-level parallelism, a combination of condition-element parallelism and join-level parallelism, and finally a combination of action-level parallelism, condition-element parallelism and join-level parallelism.

Finally, chapter 6 discusses the conclusions from the research carried out in this thesis and presents various issues related to future research work.

Chapter 2

A Survey on AI Production Systems Performance

Efficiency of production systems has been a major issue studied by many researchers. As a result, the work that has been done towards the speed-up in the execution of the Recognize-Act cycle is clearly architecture dependent. In other words, research in this area falls into two environments, uniprocessor and multiprocessor environments. In the uniprocessor environment, most efforts have been directed towards acceptable execution speed in real-time environments. Elsewhere, the focus has been the parallel execution of the Recognize-Act cycle and in particular the match phase. In this chapter, section 2.1 summarises research on uniprocessor performance, and sections 2.2 and 2.3 summarize that on shared and distributed memory multiprocessor environments, respectively.

2.1 Uniprocessor environments

Many software techniques have been investigated to speed up the execution of OPS5 based production systems and particularly in the match phase of the Recognize-Act cycle, since it is estimated that on average 90% of the time spent executing this cycle is spent in the match phase [8]. As a baseline, it is worth noting that the OPS5 inference engine in Franz Lisp runs at around 8 WM changes per second — about 3 production firings per second — on a VAX−11/780 machine.

2.1.1 Rete Match Algorithm

The Rete match algorithm represents the evolution of indexing algorithms exploited by some researchers such as McCracken, McDermott and Rychener [27, 30, 39]. Briefly, when a WM element is inserted into the WM, some features are extracted and used to generate a set of productions that are partially satisfied. Subsequently, condition elements related to each production in this set is examined to check whether a production is to be instantiated or not. In addition, memory support was exploited in these algorithms to count the number of patterns in any WM element inserted into the WM. Conversely, deletion of a WM element decreases the count by the number of patterns in the element.

The Rete match algorithm [8] can be viewed as the engine of the match phase where WM changes are inputs to this engine and productions instantiations its output. Rete can be also thought of as a data-flow network compiled from the left hand sides of the productions. Rete exploits two important characteristics of most production systems, *Temporal Redundancy* and *Structural Similarity* as described below:

(1) Temporal Redundancy: This refers to the relatively small number of WM changes that are to be executed on every production system cycle run (i.e. Recognize-Act Cycle). This entails storing results of previous production system cycles so as to use them in subsequent cycles.

(2) Structural Similarity: Often, productions exhibit structural similarities between two or more condition elements in one or more productions. This similarity is represented by similar condition elements classes and attribute-predicate-value triples in different condition elements. Exploiting such a characteristic results in sharing common tests and hence improving performance.

Building the Rete Network

One way to understand the building of the Rete network is to consider its inputs and outputs. Since a WM change is supposed to add, delete or modify a WM element, WM changes propagate through the root node from the network as tokens of the following form:

`< tag, WM data elements part >`

The tag is either + to indicate addition to WM or - to indicate deletion from WM. Hence, adding a WM element is performed by sending the following token:

`< +, WM element to be added >`

and deleting a WM element is performed by sending the following token:

`< -, WM element to be deleted >`

Modification of a WM element is performed in two steps, namely a deletion followed by an addition.

To generate the Rete network for a typical production system program, each production is analysed to identify *Intra-Condition tests* and then *Inter-Condition tests* as follows:

Intra-Condition Tests: These tests appear as direct successors of the root of the Rete network. These are called constant test nodes. These have one input and one output; in addition, the following steps are performed when compiling each condition element of a production:

1. Build a constant node corresponding to the class of each condition element if there is no node of that class immediately below the root of the network.

2. Then, build a node for each constant test attribute. As above, these nodes only need be built if no nodes exist with the same features (i.e. attribute-predicate-value) in the Intra-Condition tests portion of the Rete network.

3. Build constant test nodes for multiple occurrences of the same variable in a condition element to check for consistent variable binding within a condition element.

When a token is propagated from the root, the class of the WM element in the data elements part of the token is compared to one of the constant nodes immediately below the root node and then if one of these tests succeeds, it is sent to subsequent constant tests. If any of the constant tests fail, the token is rejected. If the token passes all the constant tests, the token is stored in a new node type called an *alpha memory* node, which holds tokens satisfying a particular condition element (i.e. satisfied all its constant

tests). This alpha memory support in Rete relieves the inference engine of computing matching WM elements relevant to different condition elements on subsequent cycles. Note that a WM element is said to be relevant to a condition element if it satisfies all its constant tests.

Inter-Condition Tests: Inter-Condition tests are performed to check for consistent variable bindings across multiple condition elements. These tests are implemented in Rete as two-input memory nodes. When a token t_i arrives at either the left or right input of a two-input memory node, it is compared with the tokens available in the opposite memory to check for consistent variable bindings. Tokens that satisfy consistent variable bindings are paired with token t_i and passed to a new memory node called a *Beta-Memory* node which may serve as a left input to another two-input node. It is worth noting that left inputs of two-input nodes are always either alpha or beta memory nodes, whereas right inputs are always alpha memory nodes. Moreover, there are two types of two-input nodes, *and-nodes* which compute consistent variable bindings between positive condition elements of the same production and *not-nodes* which compute consistent variable bindings for negated condition elements.

Finally, the result of the output of the last two-input node which holds tokens satisfying all condition elements of a specific production is stored in a new node called a *terminal node* or a *production node* having the production's name as the name of the node.

To demonstrate how OPS5's inference engine works using the Rete match algorithm, a Rete network of productions `mb1` and `mb3`, shown below, of the monkey and bananas production system program [3] is constructed. Then four WM actions are used to demonstrate how tokens are formulated and processed. Figure 2.1.1 shows how the LHS of these two productions are compiled into a corresponding Rete network.

```
(p mb1
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on ceiling)
  -->
     (make goal ^status active ^type move ^object ladder ^to <p>))
```

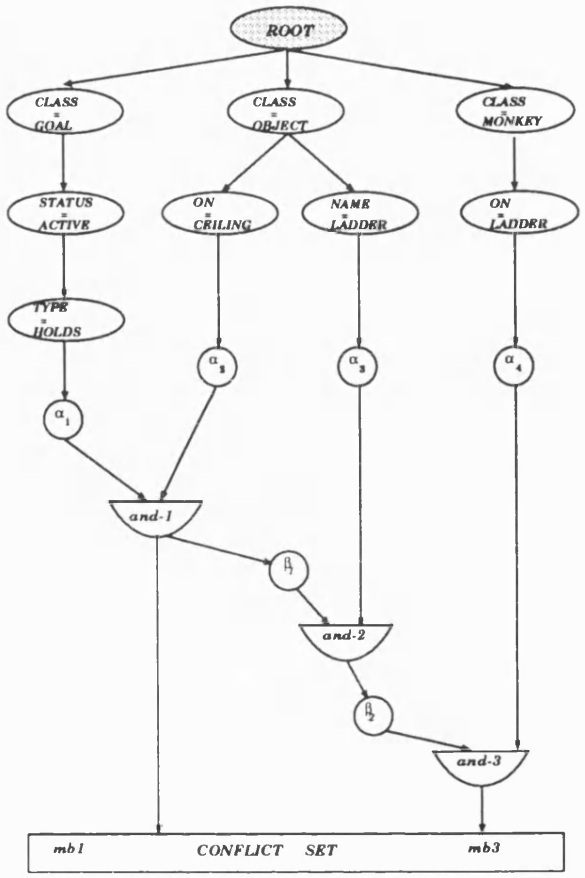

Figure 2-1: Rete Network of Productions mb1 and mb3

```
(p mb3
      (goal ^status active ^type holds ^object <w>)
      (object ^name <w> ^at <p> ^on ceiling)
      (object ^name ladder ^at <p>)
      (monkey ^on ladder)
   -->
      (make goal ^status active ^type holds ^object nil))
```

Now, consider what happens when executing the following WM actions:

```
(1): (make goal ^status active ^type holds ^object banana)
```

```
(2): (make object ^name banana ^at 5-7 ^on ceiling)
(3): (make object ^name ladder ^at 5-7)
(4): (make monkey ^on ladder)
```

For the purpose of demonstrating the Recognize-Act cycle, each of the above WM actions is assumed to require a single production system cycle. When the first WM action is executed, a new production system (PS) cycle is initiated to start the match phase. Hence, `token1` is formed as follows:

```
token1: < + , (goal ^status active ^type holds ^object banana) >
```

`token1` is then sent to the root of the Rete network from where it is broadcast to all successor nodes. All the tests fail except for `class=goal` which forwards `token1` to its successor constant test node `status=active`. `token1` passes this test and is sent to the successor constant test node `type=holds` which it passes also. `token1` is then sent to the successor node α_1 memory node which stores `token1` and passes a copy of it to the left input of `and-1`. At this stage, the right memory of `and-1` (i.e. α_2) is empty and hence processing of `token1` is suspended. This implies the termination of the current PS cycle because the conflict-set is empty.

Now, the second WM action is executed and a match phase of a subsequent PS cycle is initiated. Hence, `token2` is formed as follows:

```
token2: < + , <object ^name banana ^at 5-7 ^on ceiling) >
```

`token2` is then sent to the root of the Rete network and is broadcast to its successors. `token2` passes the constant test `class=object` and sends `token2` to `on=ceiling` which it also passes and is sent to α_2 which stores it and sends a copy to the right input of `and-1`. At this stage, `token2` is checked against all tokens in the left memory of `and-1` for consistent variable binding of variable `w`. At this moment, only `token1` exists in the left input memory of `and-1`. As a result, `token2` is checked against `token1` where the consistency check succeeds and `w` is bound to `banana`. Hence `token1` and `token2` are paired to form `token3` as follows:

```
token3:  < + , (goal ^status active ^type holds ^object banana),
             (object ^name banana ^at 5-7 ^on ceiling) >
```

token3 is then passed to both β_1 memory node and mb1 production node (i.e. terminal node). When token3 reaches mb1 node, it is sent as mb1's production instantiation to the conflict set. When token3 reaches β_1, it is stored and then a copy of it is sent to the left input of and-2. However, the right memory of and-2 is empty at this stage and hence processing of token3 at and-2 is halted.

At this stage, the conflict-set holds mb1's production instantiation, and OPS5's inference engine enters the conflict-resolution phase of the current PS cycle. Since one production instantiation exists in the conflict-set, then mb1's instantiation is selected for firing regardless of the conflict-resolution strategy being applied. As a result, a new PS cycle is initiated; consequently, a match phase is initiated and token4 is formed from the action in the RHS of mb1 production as follows:

```
token4: < + , (goal ^status active ^type move ^object ladder ^to <p>) >
```

Then, token4 is passed to the root of the Rete network of productions mb1 and mb3 in Figure 2-1. Although it passes the tests class=goal and status=active, it fails at type=holds and processing halts ending the current PS cycle.

When the third WM action is executed, a new PS cycle is initiated and token5 is formed as follows:

```
token5: < + , (object ^name ladder ^at 5-7) >
```

token5 satisfies class=object and name=ladder and hence it is sent to α_3 where it is stored and a copy is sent to the right input of and-2. token5 is now compared against all tokens in the left memory of and-2, in this case only token3, from an earlier cycle, for a consistent variable binding of variable p. As a result, p is bound to 5-7 and token5 and token3 are paired to form token6 as follows:

```
token6: < + , (goal ^status active ^type holds ^object banana),
             (object ^name banana ^at 5-7 ^on ceiling)
             (object ^name ladder ^at 5-7) >
```

token6 is then sent to β_2 beta memory which stores it and sends a copy to the left input of and-3 where processing of token6 is terminated because and-3's right input is empty.

The fourth WM action is executed and token7 is formed:

```
token7: < + , (monkey ^on ladder) >
```

This passes class=monkey and on=ladder, reaching α_4 where it is stored a copy is passed to and-3's right input. token7 is paired with all tokens of the left input memory of and-3 (token6) since no consistent variable bindings are to be computed at this node and hence token8 is formed and sent to mb3 production node.

```
token8: < + , (goal ^status active ^type holds ^object banana)
              (object ^name banana ^at 5-7 ^on ceiling)
              (object ^name ladder ^at 5-7)
              (monkey ^on ladder) >
```

Production node mb3 sends token8 to the conflict set as an instantiation of mb3. Then, the conflict-resolution phase is started, where mb3 production is selected and fired. token9 is formulated from the action in RHS of mb3 as follows:

```
token9: < + , (goal ^status active ^type holds ^object nil) >
```

This passes class=goal, status=active and type=holds and reaches α_1 where it is stored and passes a copy to the left input of and-1 for consistent variable bindings with tokens in the right memory of and-1. Consistent variable binding test fails at and-1 since its right memory, α_2, has one token, token2 with variable w bound to banana while it is bound to nil in token9 of α_1. As a result, processing of token9 is terminated.

The inference engine now stops because there are no more WM actions to be executed and the conflict-set is empty.

2.1.2 OPS5 based on BLISS

OPS5/LISP was followed by OPS5/BLISS [11] in which OPS5 was rewritten in a systems programming language, called BLISS. Performance was about 5 times faster than

OPS5/LISP. This speed-up was a result of changing the representation of the important data structures and adding special code to handle common cases efficiently.

2.1.3 OPS83

In another attempt to improve the performance of OPS interpreters, OPS83 was developed. In OPS83, left hand sides of productions are compiled into native machine code. OPS83 was about 4 times faster than its predecessor OPS5/BLISS [11] giving around 200 WM changes per second.

2.1.4 The TREAT Algorithm

TREAT [31] stands for Temporally REdundant Associative Tree algorithm and was initially developed to run on the DADO machine [43], details of which will be given later. TREAT was originally intended as a parallel algorithm but a serial algorithm was derived later for temporally and non-temporally redundant production systems. In addition to supporting condition membership and memory support, TREAT utilises a conjecture made in McDermott et. al [30] with respect to conflict-set support:

> "*It seems highly likely that for many production systems, the retesting cost will be less than the cost of maintaining the network of sufficient tests.*"

TREAT further utilises conflict-set support by using it as a tool to reduce the number of comparisons required to compute consistent variable bindings. To exploit conflict-set support, a distinction must be made between processing positive and negative condition elements.

`Processing Positive Condition Elements:` Adding WM elements that match positive condition elements may result in new production instantiations which are added to the conflict set. Deleting a WM element that matches a positive condition element is handled by removing the resulting production instantiations including that WM element from the conflict set directly without the need to compute consistent variable bindings as in Rete.

Table 2.1: Speed-up obtained by TREAT over Rete

Order	MAB	MUD	MESGEN	MAPPER	WALTZ
Lexical	1.48	0.77	2.22	1.14	0.65
Sorted	1.67	1.96	2.22	1.14	1.48

`Processing Negative Condition Elements:` Adding WM elements which match negative condition elements may result in removing some productions instantiations from the conflict set. However, TREAT handles this case by temporarily transforming a negative condition element into a positive one and then removing productions instantiations from the conflict set that would be added as a result of this transformation. On the other hand, removing a WM element is handled in the same way as if the condition element had been positive and the WM element had been just added.

In contrast to Rete, while TREAT computes constant tests and stores matching tokens in memory nodes like Rete, it partitions alpha memories into *old*, *new-add* and *new-delete* which hold WM tokens of previous cycles, new WM elements to be added and new WM elements to be deleted, respectively. Moreover, TREAT does not build beta memories nor does it store computations of consistent variable bindings. Instead, TREAT utilises a database approach in joining relations with respect to their increasing cardinality[1] [46]. In summary, a WM element matching a specific condition element is used as a seed to initiate the join procedure between different alpha memories of one production ordered with respect to increasing cardinality. Table 2.1[2] shows the speed-up obtained by TREAT over Rete using both the lexical ordering and seed ordering join of alpha memories in a number of programs.

2.1.5 RISCF Production System Machine:

Forgy [8] originated the idea of exploiting RISC (i.e. Reduced Instruction Set) [35] architectures to implement the Rete match algorithm which was later studied by Lehr [24]. The main motive behind exploiting RISC architectures in executing production

[1] Refers to number of tuples in a relation
[2] Figures in this table are extracted from the graph in Figure 4-11 in [31]

systems using Rete-based OPS5 interpreters is that the Rete match algorithm does not require arithmetic operations nor sophisticated addressing modes. Typical instructions performed by Rete when compiled into assembly language are load, compare and jump.

Rete RISC, named RISCF, was a special purpose architecture to execute Rete based OPS interpreters and has five classes of instructions: jump, subroutine calls, memory reference, immediate and register to register classes, where each instruction is encoded in 32 bits. RISCF has also four addressing modes, absolute, indirect, base register plus displacement and register.

A study of applying branch and prediction strategy of RISCF to Rete was conducted by Lehr [24]. The branch and prediction strategy in RISCF relies on two bits in the instruction one of which is used to predict a branch and the other one to inform the processor that the next instruction is jump class. So if a branch is taken one machine cycle is saved.

Since the success of the branch and prediction strategy depends heavily on the amount of available branching in a typical program, Rete-based production system programs are expected to gain when such a strategy is utilised because constant test nodes, memory nodes, and-nodes and not-nodes will probably lead to branching. As a result the gain in performance when utilising such a strategy varies from one OPS production system to another. Lehr [24], concluded that for the six OPS5 production systems studied in [12], the average speed-up obtained when utilising such a strategy at constant-nodes, memory nodes, and-nodes and not-nodes is 15% based on run time measurements obtained by Gupta and Forgy [12].

2.1.6 Comparative Analysis Of Computer Architectures

A survey was conducted by Quinlan [36] to improve production systems performance by means of comparing different computer architectures. The following six diverse computer architectures were investigated:

1. **A custom micro-coded machine:** The two main features of this architecture are the large general purpose register file and the fact that each instruction represents an operation on either an entire node or large part of a node.

2. **RISCF Production System Machine:** RISCF is a custom-designed RISC machine for executing Rete based OPS production system programs as described in section 2.1.5.

3. **RISC II Computer:** This machine features a reduced instruction set that resides on a single chip.

4. **The VAX-11/780:** This machine is the basis of many of the studies that have been conducted to improve OPS production systems performance. This machine features a large instruction set where instructions are of variable length format, but only a small number of registers.

5. **The Pyramid 90x Computer:** This machine features a large instruction set of fixed format.

6. **A Computational Model:** A computational model was developed to establish an upper bound on the execution of production system programs based on the following metrics when executing a typical program:

   (a) The minimum amount of information required to do an operation and hence calculate the number of instruction bytes read from primary memory.

   (b) The number of bytes of data transferred between the processor and main memory.

   (c) The number of computations required of the processor such as ALU operations, data address calculations and branch address calculations.

The results of the survey conducted on these six architectures were based on results obtained by Gupta and Forgy [12] regarding six production systems and can be summarised as follows:

1. On average, the micro-coded machine requires one half of the CPU-Memory traffic required of the others due to the large register file.

2. On average, the RISCF machine required the lowest number of machine cycles due to the branch and prediction strategy exploited in this architecture. RISCF machine comes second to the micro-coded machine for CPU-Memory traffic.

3. The use of hash tables to store tokens instead of linked lists as in OPS5, resulted in a reduction of CPU-Memory traffic by between 0.75 and 5.88 for the six sample inputs. This shows that an increase or decrease in performance is program dependent if hash tables are to be used to store tokens instead of linked lists. Moreover, using hash tables, the study showed that adding and deleting WM tokens takes 50-80% of the total CPU-Memory traffic independent of the production system being used.

4. The computational capabilities of a CPU executing production systems are mainly data loading and storing.

As a result, Quinlan [36] concluded that the micro-coded machine is the best CPU architecture in this survey to execute production system programs due to resulting heavy CPU-Memory traffic. On the other hand, the RISCF machine may be a good candidate if a cache memory is to be incorporated to reduce CPU-Memory traffic and provided the cycle time is small enough to rival that of the micro-coded machine.

2.2 Shared Memory Multiprocessor

In an attempt to improve the performance of production systems, Gupta [15] in his Ph.D. studied the parallelism available in OPS5 production systems that are based on Rete. The results of this study formed the basis of the strategies behind the design of the Production System Machine (PSM). First, a summary of the results of parallelism in production systems is presented in section 2.2.1 and then the architecture of the Production System Machine (PSM) is presented in section 2.2.2.

2.2.1 Parallelism in Production Systems

The processing done in the match phase can be broken into two parts [33]: selection and state-update. The first involves obtaining the set of condition elements that are satisfied by a new WM change (i.e. the task of performing intra-condition tests in Rete). The state-update part is concerned with obtaining new production instantiations as a result of updating state associated with satisfied condition elements in the selection part and then performing inter-condition tests.

Gupta studied parallelism in the match phase with respect to state-update based on run-time statistics for six-well known production systems, VT [26], ILOG [29], MUD [20] , DAA [22], R1-SOAR [38] and EP-SOAR [23]. These showed 75-95% of the processing time is spent in performing the state-update part. Gupta studied the following kinds of parallelism within state-update:

Production-Level Parallelism

Production-level parallelism entails partitioning a production system program into several partitions of productions, where in the extreme case each production is assigned to its own processor. Simulation results obtained from production-level parallelism show that the speed-up obtained seems bounded by a factor of 1.9. Gupta attributed this limited speed-up to the following reasons:

1. The average number of affected productions per WM change is small compared to number of productions in a typical production system. However, for the six production systems mentioned above, the average size of the affected set is 26. This will result in underutilisation of the processors in the machine being used if the number of processors is much larger than the size of the affected set.

2. The variance in processing requirements of the affected set of productions.

3. The loss of sharing in the Rete network.

Node-Level Parallelism

In an attempt to reduce the variance in processing requirements of the affected set of productions as explained above, node-level parallelism was explored. Node-level parallelism refers to the processing of activations of two-input nodes in parallel regardless of whether the two-input nodes belong to the same production or different productions in the Rete network. Simulation results indicate that, on average, a speed-up of 1.5 may be obtained over production-level parallelism if node-level parallelism is used. Since node-level parallelism requires a massive amount of communication, a shared memory architecture is preferred.

Gupta noted two disadvantages of node-level parallelism, known as the *long chain* and the *cross product* effects. The long chain effect is characterised by a long chain of dependent two-input nodes. Productions having such a characteristic may take a long time to process the activation of such a chain of two-input nodes. Therefore, this may result in variance in processing of affected productions. To resolve this, Gupta proposed the use of binary Rete network. Using such a technique, simulation results showed a a gain in performance for SOAR but a loss for others. The gain for SOAR OPS-like programs is due to large number of condition elements in productions of such programs. Alternatively, the loss in performance in other OPS programs is due to the small number of condition elements in productions of such programs because the cost of updating state in the binary Rete network is much higher than in the linear Rete network. The cross product effect refers to the situation when an incoming token to one input of a two-input node is to be matched with large number of tokens in the opposite memory. As a result, a large number of tokens may be sent to the successor nodes which are processed sequentially. Hence, productions suffering from this effect take more time to be processed and affect the overall speed-up. To resolve this, Gupta proposed intra-node-level parallelism described below.

Intra-Node Level Parallelism

Intra-node level parallelism is the processing of multiple activations of two-input nodes in parallel in an attempt to reduce the impact of the cross product effect described

above. Simulation results obtained by Gupta show that on average a speed-up of 1.3 may be obtained when using intra-node level parallelism over node-level parallelism. However, the use of intra-node level parallelism may lead to incorrect results in two cases as follows:

- When the opposite memory of a two-input node is unstable. That is, processing of deletions or additions of tokens in the opposite memory while processing activations from the other memory of the same two-input node.
- Processing of conjugate tokens in parallel. This refers to a situation where the same token being inserted is also being deleted.

Gupta suggested sequential processing of the first case because of the assumption in the Rete network that while a two-input node is being processed, the opposite memory should stay stable. He also noted that there is no simple way to ensure that the opposite memory stay stable and it would be expensive to detect and delete duplicates leaving two-input nodes[3]. In the second case, Gupta proposed the use of an extra-deletes-list to store earlier deletes associated with two-input nodes so as when a new token is inserted, a check is first made whether there was an earlier request for its deletion or not. Gupta did not comment on maintaining this list from cycle to cycle as this is expected to grow significantly during the life of a production system (i.e. during its execution).

Action-Level Parallelism

Action-level parallelism refers to the concurrent processing of WM changes. Simulations showed that speed-up factors of 1.5, 1.85 and 3.9 may be obtained over production-level parallelism, node-level parallelism and intra-node level parallelism, respectively when combined with action level parallelism. Hence, the upper bound on the speed-up obtained from using parallelism is around a factor of 10 over uniprocessor Rete-based OPS interpreters.

[3] more elaboration on this case is presented in section 5.2.4

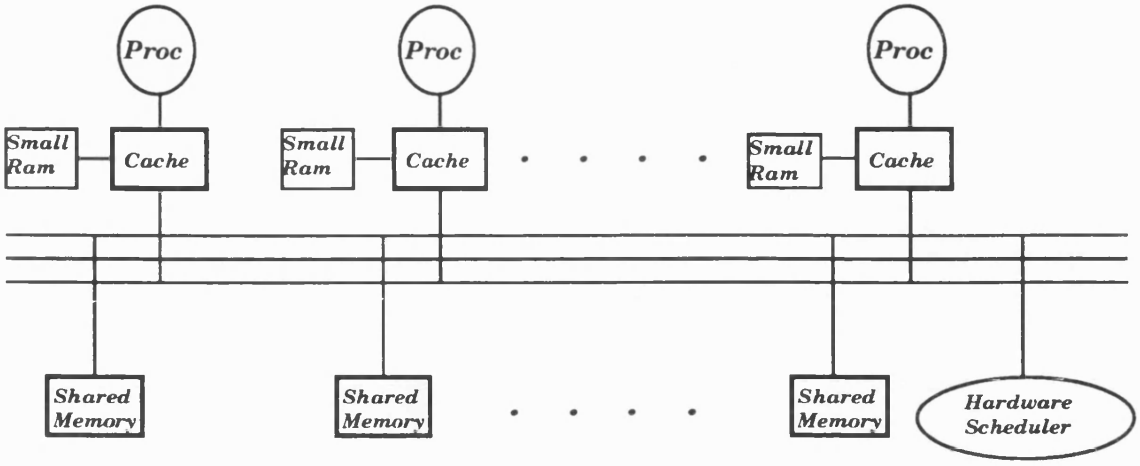

Figure 2-2: Architecture of the Production System Machine

2.2.2 Production System Machine

The Production System Machine (PSM) is a result of the research carried out by Gupta [15] in his Ph.D. to study parallelism as discussed above. PSM is a shared-memory multiprocessor architecture as shown in Figure 2-2 with the following characteristics:

- The number of processors is chosen between 32 and 64 based on the simulation results obtained when production-level parallelism was explored [15], which showed that the size of the affected set of productions is relatively small. Also, his simulations showed that using more than 32 processors does not produce an additional speed-up.

- Individual processors are high performance RISC processors each with a small amount of RAM and cache memory. The use of RISC processors is based on results obtained by Quinlan in section 2.1.5.

- Processors should be connected to shared memory by one or more shared buses. However, simulations obtained by Gupta show that a single high speed bus is capable of dealing with the load put on it by 32 processors.

- A hardware task scheduler is to be connected to the shared buses to schedule processing of different node activations as well as assigning such activations to

idle processors. This scheduler has to be very fast so as not to be a bottleneck bearing in mind that node activations are very small tasks.

The suggestion to use shared memory architecture is an attempt to solve two problems: first, when adapting the Rete network to a multiprocessor system using node-level or intra-node level parallelism there is huge amount of communication between nodes; second, to simplify the load distribution problem which would result in a non-shared memory architecture when, for example, assigning node activations to different processors. The assignment of productions to processors in the general case is shown to be NP-complete [33].

The estimated performance of the PSM machine when running the adapted Rete algorithm is 11,000 WM changes per second using 32 processors each rated at 2 MIPS.

2.3 Non-Shared Multiprocessor Environments

Research towards improving the performance of production systems has led some researchers to propose specialised distributed memory architectures and the following is a survey of that research.

2.3.1 OPS5 Production Systems on ILLIAC-IV

ILLIAC-IV is a parallel processor consisting of 64 processing elements (PE) running in SIMD (Single Instruction Multiple Data stream) mode. Forgy [9], described an algorithm to exploit these SIMD PEs in executing production systems. This algorithm entails that a production system be partitioned into 64 sub-production systems corresponding to each PE. Each PE is assigned one of these partitions and constructs its own Rete network for that set of productions. The execution of the match phase is constrained by the SIMD nature of this machine such that no processor starts processing nodes of further types unless all other processors finish processing current node type. In other words, constant-test nodes are processed first by all processors and then alpha memory nodes are to be processed next and so on. Due to this constraint, PEs which finish early stay idle till the last processor finishes processing the current evaluation of a

particular node type. This constraint has been reported as the most serious limitation and it may be concluded that SIMD machines are not good candidates for executing production systems especially Rete-based ones. It has also been proposed that productions that possess structural similarity be put in one partition as a one remedy to minimise the time some processors stay idle. However, this is a static solution to load balancing and does not consider the dynamic problem. Finally, no estimates have been reported with respect to speed-up expected from this approach.

2.3.2 The DADO machine

DADO is a special-purpose machine designed to achieve high performance in executing rule-based systems based on a very large scale parallel architecture [43]. DADO has a binary tree topology and is comprised of very large number of processing elements(PEs). Each PE has its own processor, memory and a specialised I/O switch.

This flexible architecture supports MSIMD (Multiple Single Instruction Multiple Data streams) processing mode. Each PE is able to operate in two modes, SIMD and MIMD(Multiple Instruction Multiple Data streams). A PE in SIMD mode receives instructions from some ancestor PE but operates on different sets of data residing in its own local memory. On the other hand, a PE in MIMD mode executes instructions in its local memory independent of other PEs in the tree. The root of the tree is a single processor that controls the entire operation of the tree. In practice, the DADO machine can be configured in such a way so that one or more PEs become the root(s) of one or more subtree(s) in the tree, where the root of a subtree operates in MIMD mode and PEs of this subtree operate in SIMD mode. As a result, the DADO machine can be divided into logically distinct partitions, where each partition executes a certain task. Communication between physically adjacent PEs (i.e. parent and children) is handled by a specialised I/O switch that contains a combinatorial circuit to manage rapid selection of a candidate PE in the conflict-resolution phase of the Recognize-Act cycle.

A number of algorithms have been proposed to exploit DADO's binary topology and parallel architecture in executing production systems. The following is a summary of

these algorithms.

Algorithm 1: Full Distribution of Production Memory

In this algorithm, a production system program is to be partitioned into an equal number of partitions corresponding to the number of PEs in the DADO machine. Each PE has its own set of productions, a naive match algorithm and a set of WM elements relevant[4] to its set of productions. WM changes are broadcast to all PEs and hence each PE computes match and conflict-resolution. Selection of a winning production among all PEs is performed by the max-resolve circuit of the DADO machine. As stated in [43], the performance of this algorithm depends on the nature of the match algorithm being used. However, since the local memory of each PE is quite small (e.g. 16KB), the use of a sophisticated match algorithm such as Rete [8] may cause memory contention problems and this will be exacerbated when the size of a PE's local WM is large.

Algorithm 2: Original DADO Algorithm

In this algorithm, the DADO machine is logically divided into three levels: Production Memory(PM) level, upper tree level and WM level subtrees as shown in Figure 2-3. The selection of the PM-level is based on any of the following two strategies. One is based on assigning one PE per production in the PM-level and thus selecting a PM-level that has a number of PEs that are at least equal to the number of productions in PM. The other is based on subdividing the PM and distributing it to the PEs in the PM-level. Thus, selecting the PM-level is based on finding a level in the tree that has a minimum number of PEs equal to the number of partitions of a typical production system. Each of the PM-level PEs operates in MIMD mode while their successors constitute WM-subtrees level and operate in SIMD mode.

Processing starts by broadcasting WM changes to each PE in the PM-level and this concurrently determines the relevancy of the WM-changes broadcast. If a WM change specifies addition, it is added to a free PE in a subtree rooted at a PM-level

[4] A working memory element is said to be relevant to a production if it satisfies all constant tests of one of the condition elements that production.

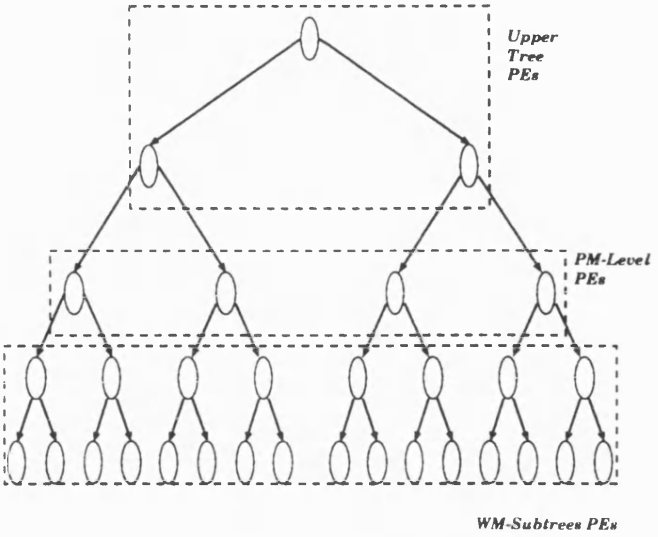

Figure 2-3: Application of DADO Original Algorithm

PE. On the other hand, deleting a WM change is performed associatively using the content addressable hardware of DADO. Each condition element of a production stored in PM-level PEs is broadcast to the WM-subtree below its PM-level PE for consistent variable binding testing. Local conflict sets of productions are formed and stored within relevant WM-subtrees. Upon termination of the match phase and starting of the select phase, PM-level PEs synchronise with the upper-tree level and conflict-resolution is performed using the max-resolve circuit of DADO. Resulting production instantiations are reported to the root of the tree to be fired and resulting WM changes are broadcast to the PM-level PEs to start a new cycle.

This algorithm does not seem suited for temporally redundant production system programs (i.e. many WM changes are to be executed on every production system cycle) because of the recomputation of the match results done in the previous cycles. In addition, a large number of changes to WM may cause memory contention because of the small size of the PE RAM and particularly in asynchronous production systems [40], where WM changes are input both from external sensors and as a result of firing productions. No details regarding implementation or performance of this algorithm has been seen so far.

Algorithm 3: TREAT

The TREAT algorithm described previously for a uniprocessor was initially proposed to run on DADO. In the full distribution algorithm, the match routine is replaced by TREAT on each PE. On the other hand, TREAT was a refinement of the original DADO algorithm with respect to state saving and particullarly the construction of alpha memory nodes (recall that the DADO original algorithm does not save state between production system cycles).

TREAT has been implemented (in C) with the full distribution algorithm. The program fits into 8KB of RAM leaving 8KB for productions and WM elements (each DADO node has only 16KB of RAM). Testing with a sample of four production systems showed it was up to 7 times faster than Rete. Miranker [31] also estimated the speed-up obtainable from TREAT with PM-level distribution to be 3 to 14 times faster than Rete. He did this by dividing the speed-up from TREAT with full-distribution by that from partitioning production memory across the PM-level PEs. Miranker concluded that a DADO machine of 32 to 127 processors, each with larger memories would be a better configuration for the DADO architecture because current PE memory is too small to store large numbers of WM elements and because production systems have limited amount of parallelism. This conclusion confirms one of the intuitions behind the design of the Production System Machine despite the fact that both PSM and DADO differ in both architecture and topology.

Algorithm 4: Fine-grain Rete algorithm

Under this scheme, Rete is mapped on to DADO from the bottom-up. Hence, DADO is divided into three levels as in the original DADO algorithm but the upper tree which represents the leaves of the DADO binary tree is responsible for computing constant tests and storing alpha memory tokens. The PM-level stores production nodes (i.e. terminal nodes) of the Rete network. The levels of the tree close to the root are responsible for computing conflict resolution. As a result, a winning production is propagated to the root and then fired. WM changes that result from firing are broadcast to all PEs in the upper tree.

This suggested implementation of Rete on DADO was studied by Gupta [14] who suggested the following:

- The sharing feature in Rete match algorithm should be turned off.
- Constant test nodes leading to an alpha memory node are to be stored in the same PE.
- Memory nodes that are inputs to two-input nodes are to be stored in the same PE along with their respective two-input node so as to reduce communication overhead when processing two-input nodes.
- The binary tree structure of DADO is to be exploited instead of the linear structure of Rete networks. This approach resembles that taken to solve the long chain effect in node level parallelism as was explained in section 2.2.1.

Gupta analysed the performance of this implementation and estimated 12 production firings per second. Although this algorithm seems to be better than the original DADO algorithm, it still suffers from some problems such as the limited size of PE memory and the performance of production systems running this algorithm may be governed by the variance in processing requirements of some productions.

Algorithm 5: Using DADO as Associative Processors

An attempt was made by Gupta [14] to respond to the deficiencies of DADO's fine-grain Rete algorithm by partitioning a production system into K partitions equal to the number of PEs in a particular level of the DADO tree chosen as the PM-level. The selection of the value of K depends on production level parallelism analysis as shown in section 2.2.1 which showed that on average the size of the affect set is 26. As a result, a separate Rete network for each of the K partitions is to be constructed. However, the performance of this algorithm depends on the success of the partitioning strategy and whether PE memories are large enough to contain sufficient WM elements. The performance of this algorithm was estimated to be 67 production firings per second.

This algorithm takes advantage of DADO's binary tree topology, which also allows division into sub-machines, each rooted by a single PE. Although DADO has a large scale parallel architecture, it certainly suffers some drawbacks: (1) PEs used are not powerful enough (2) PE memory is too small to hold sophisticated match routines such as Rete or others. (3) Underutilisation of PEs, in a sense that a large number of PEs may stay idle during the match phase due to variance in processing requirements of productions.

In conclusion, we observe that although there have been many attempts at partitioning the problem to fit the DADO architecture, this remains a problem for which an optimal solution is yet to be found.

2.3.3 NON-VON Machine

NON-VON was designed at Columbia University [17] and is a highly parallel tree structured architecture designed initially for applications in symbolic information processing. NON-VON is comprised of two basic parts: primary processing subsystem and secondary processing subsystem with a front end connection to a host as depicted in Figure 2-4. The primary processing subsystem has a binary tree structure consisting of a very large number of small processing elements (SPEs) that operate in SIMD mode. Each SPE has an 8 bit ALU and a very small RAM in the range of 32 to 256 bytes and communication links to its right and left neighbours via its parent. Leaf SPE nodes are interconnected to form a two-dimensional orthogonal mesh. Large processing elements (LPEs) operating in multiple-SIMD mode, are supposed to root the SPEs at the fifth to the tenth level of the primary processing subsystem. The secondary processing subsystem consists of a number of disc drives where each one is connected to one LPE in the primary processing subsystem via an intelligent head unit.

An algorithm was proposed for executing OPS5 on NON-VON which appears to be a modification of Gupta's algorithm "Using DADO as a Multiple Associative Processors". The proposed algorithm is to run on a prototype consisting of 32 LPEs corresponding to the number of partitions recommended by Gupta for production level parallelism as shown in section 2.2.1, and 16,000 SPEs. Each LPE receives a subset of productions

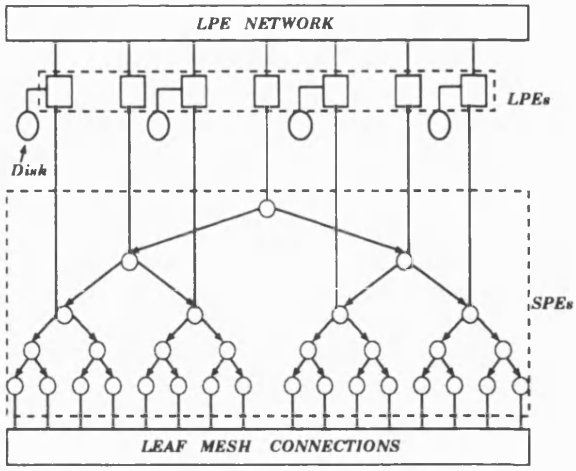

Figure 2-4: Architecture of the NON-VON Machine

and builds a corresponding Rete network but contents of memory nodes are stored in the many SPEs of which the LPE is the root. Since the memory of the SPE is very small, a WM token is processed by many SPEs at the same time. The host processor is responsible for both executing both the select and act phases.

The estimated performance of this algorithm for this prototype using 3 MIPS LPEs (Motorola 68020) with a program of 910 productions is 850 production firings per second.

Clearly, the distinction between DADO machine and NON-VON machine with respect to performance lies in the very powerful LPEs processors and the massive number of SPEs rooted by each LPE. However, it may be concluded that NON-VON machine is a better candidate for executing very large production systems employing large numbers of WM changes.

2.3.4 Oflazer's Work

Oflazer in his Ph.D. thesis [33] investigated some issues related to parallel processing of production systems and particularly OPS based ones. The main issues investigated are: (1) Implementation of an OPS5 parallel interpreter (2) Partitioning of productions (3) A new parallel algorithm for production systems execution (4) A new parallel architecture to support this algorithm. The following summaries the major results obtained,

Implementation of a Parallel OPS5 Interpreter

A parallel OPS5 interpreter was developed by Oflazer to run on a VAX−11/784 multiprocessor system comprising four processors connected to a large shared memory. This interpreter exploits production level parallelism [15] and hence a production system program which is executed using this interpreter ought to be partitioned among the four processors. In general, the structure of such an interpreter reduces to a set of heavyweight processes derived from a functional partitioning of the Recognize-Act cycle. In this implementation, processors communicate using UNIX operating system kernel facilities. More precisely, this interpreter is comprised of one control process and N match processes for a multiprocessor system comprised of $(N+1)$ processors. The control process is responsible for interfacing with the user, initiating the match processes, receiving partial conflict set entries from the match processes, computing final conflict resolution and then sending WM changes of the winning production, if any, to the match processes. On the other hand, each of the match processes is responsible for performing match on the subset of productions assigned to it, computing local conflict resolution and then sending the local winning production to the control process. Implementation results of this interpreter resulted in a speed-up factor of 1.1 to 1.7 for some production systems and loss in performance for some others compared to Forgy's OPS5 interpreter [10]. The loss in performance in some production systems and the little speed-up in some others can be attributed to the following reasons:

- Variance in processing requirements of productions in the affect set as a result of exploiting production-level parallelism.
- Communication overhead due to the fact that it is handled through UNIX kernel instead of a hardware bus.
- Sequential transmission of WM changes from the control process to the match processes instead of by broadcast.

Partitioning of Production Systems

The partitioning problem related to production systems can be defined as:

"*Given a set of K processors, each allocated its own memory, and a production system comprised of N productions, find a method to assign the N productions to the K processors such that the load on these K processors is uniformly distributed.*"

In selecting a partitioning strategy for production systems, one has to consider two matters that are inherent in production systems. First, state of production systems changes from one cycle to the next. Second, processing requirements of productions vary.

In studying the partitioning problem, Oflazer noted the following strategies:

1. **Random Assignment:** Self explanatory!

2. **Round-Robin Assignment:** assign productions to processors in a round robin manner such that the ith production is to be assigned to processor $(i \ mod \ (k+1))$, where k is the number of processors. The essence behind this type of assignment is that productions that are close to each other in a production system program are expected to interact with each other more often than others. Hence, such an assignment may help if production systems are written with this in mind.

3. **Context-based Assignment:** assign productions with related contexts in a round robin manner based on the assumption that productions with similar contexts can be active at the same time when processing a WM change having the same context (i.e. class of WM change is the same as class of some condition elements of some productions). However, this assumption may not help since production systems possess structural similarity and such partitioning will depend on a small number of classes.

These partitioning methods have one common characteristic in that they are all static. Later, Oflazer proposed a new heuristic method based on run time behaviour of production systems and proposed an algorithm to support it. This algorithm accepts the following inputs:

1. Number of productions in a typical production system.

2. Number of processors (or number of partitions)

3. The affect set[5].

4. The processing cost of each production in the affect set to be gathered in a particular run.

The performance of this algorithm was 1.1 to 1.25 times faster when compared to other partitioning strategies. Additionally, Oflazer recast the partitioning problem as a minimisation problem and showed that it was NP-complete.

The highly-parallel state processing algorithm

In an attempt to reduce variance in processing requirements of affected productions resulting from exploiting production level parallelism, Oflazer proposed a new highly parallel algorithm for the state update part[6] of the match phase. This algorithm is based on the strategy of storing tokens that match all the condition elements of a particular production rather than a fixed combination as in Rete. To support such an assumption, Oflazer suggested that the state of a production is represented by a set of Instance Elements (IEs), each having the following structure:

$< (t_1, w_1), (t_2, w_2), ..., (t_i, w_i), ..., (t_c, w_c) >$

where

c is the number of condition elements in a production.

t_i is the tag associated with working memory w_i matching condition element i to help in controlling the generation of redundant IEs.

w_i is WM element relevant to condition element i

In addition a special WM element X is introduced as a null WM element that satisfies all condition elements.

[5]Recall that the affect set refers to the set of productions affected by WM change in a particular cycle

[6]Oflazer divided the match phase into two parts: select and state update. In the select part, condition elements that are satisfied by a WM change are determined. In the update state part, the state of the condition elements obtained in the select part is updated.

This algorithm works in two steps. First, Instance Elements corresponding to each production are processed concurrently on an available set of processors. This implies another level of parallelism in addition to production level parallelism. Second, Instance Elements modified or generated as a result of processing performed in the first step are checked for redundancy sequentially. The need for a redundancy check stems from the fact that redundant instance elements produce incorrect match results when deleting WM elements unless they are removed.

This algorithm suffers from three potential problems. First, the state associated with some productions may quickly grow very large and hence a large number of processors will be required to update its state in order for these productions not to be a bottleneck. As a solution, Oflazer suggested splitting such productions into two or more productions where one calls the others in a specific order by sending messages upon firing. However, this solution has some disadvantages: (1) it enforces sequential processing of such productions: (2) it creates extra overhead in splitting productions both at run time and for the production system programmer; (3) it splits knowledge between several productions which is contrary to the production system philosophy of one rule for each chunk of knowledge. Second, the redundancy check being a sequential process hampers the parallelism attained from the concurrent processing of instance elements. Third, Gupta [15] reported that exploiting action level parallelism in this algorithm is difficult because it requires sequential processing of multiple changes to the slots of instance elements of a production. As a result, multiple changes to WM which may affect the same set of productions entail their instance elements be able to maintain several changes to their slots.

Finally, this algorithm does not seem to be appropriate for production systems that have one or more of the following features:

- Productions with a large number of condition elements.
- A WM element being inserted or modified matches two or more condition elements of the same production at the same time.
- Intra-condition tests among condition elements are not sufficiently selective lead-

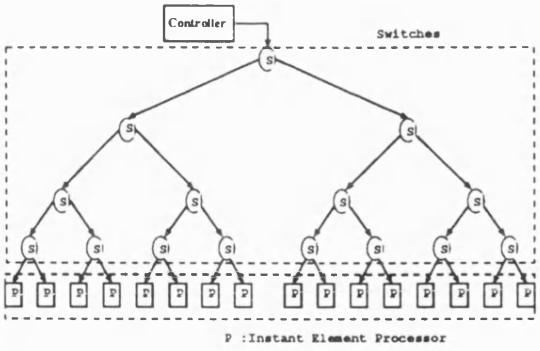

Figure 2-5: Oflazer's Parallel Processor System

ing to a growing production state (i.e. large number of IEs).

The Parallel Architecture

Oflazer proposed a non-shared memory tree-structured architecture to execute the parallel algorithm presented above. Processors are located at the leaves of the tree, where each processor has its own memory and operates in MIMD mode. Switches, which are simple processors, each having a small ALU, control unit and three bidirectional communication paths, are placed at the interior nodes to constitute a network that connects processors at the leaves to the front-end controller processor and other leaf processors as shown in Figure 2-5. Oflazer suggested the following mapping of PS programs to this proposed parallel machine in implementing the parallel state algorithm:

1. Every single production is to be assigned to a subset of processors. Hence, each leaf processor is allocated a subset of productions in a typical PS program.

2. As a result, the task of each processor is to process IEs related to a subset of productions and update the state of their respective IEs.

3. The interior switches serve as bidirectional data paths. First, they are responsible for bidirectional communication during the redundancy check phase. Second, they are responsible for sending productions instantiations to the front-end controller processor. Third, they are responsible for broadcasting WM changes from the front-end controller processor to the leaf nodes.

Oflazer used four PS programs to simulate the execution of the parallel state algorithm on his proposed parallel architecture using a simulator built on top of a Lisp-based interpreter running on a VAX-11/780. The results of this simulation showed that using 239-308 processors, it is possible to get a speed of 2200-7000 WM actions per second[7] However, despite these performance figures, it can be easily shown that on average, the redundancy check time utilises 36-77% of the total time spent in processing IEs state per cycle[8].

2.3.5 PESA-1 Production System Machine

PESA-1 [37], an acronym for Parallel Expert System Architecture, is a dedicated parallel architecture proposed for concurrent execution of production systems. PESA-1 possesses a bus-based, pipelined and dynamic data flow architecture. The basic idea behind the design of PESA-1 is to map the Rete discrimination network into a physical structure. However, this mapping is different from other approaches in that it partitions the Rete network into logical levels. Then, each of these logical levels is to be mapped into a physical level in PESA-1. A logical level in a Rete network represents a set of alpha and beta memory nodes. To support such a mapping, the following architecture depicted in Figure 2-6 was proposed:

- Four physical levels are to be used in the mapping process. However, no justification is given for the selection of four levels. It is probably because there are four to five condition elements per production on average and hence three to four joins are required on average.
- Level 1 to represent the host and each of the levels 2, 3 and 4 has many processing elements.
- Five busses are used to interconnect the PEs in the three levels and the host as shown in Figure 2-6, such that a PE at level i gets its input from bus i and its

[7]This implies that for a certain PS program, a certain number of processors is allocated to it depending on the number of IEs to be processed every cycle and hence a certain speed was obtained.

[8]This redundancy check ratio's range was calculated by the author based on figures in table 8-7 of Oflazer's Ph.D. thesis.

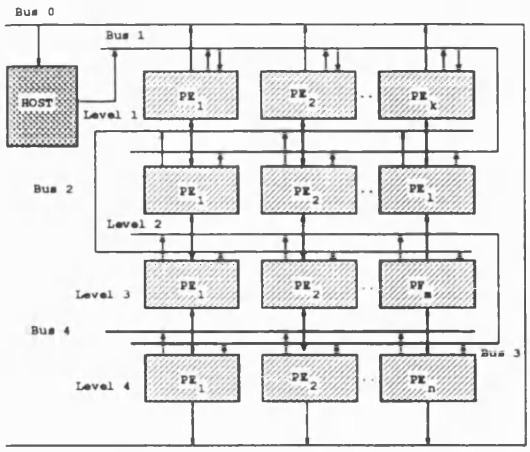

Figure 2-6: PESA-1 Production System Machine

output may be sent to any of the buses, $(i-1)$, i, $(i+1)$ $mod5$

PESA-1 Distributed Rete Algorithm

This algorithm can be divided into two parts: compile-time and run-time. The first generates the Rete network, partitions it into levels and then processes allocated to each partition are assigned to each PE in that level. At run-time, WM changes are generated asynchronously as tokens by the host. At level 1, each PE computes intra-condition tests which correspond to constant tests stored in the PE's local memory. If they are successful, tokens are sent to the levels below.

When a token is received at bus i such that $i > 1$, it is broadcast to all PEs on that level such that each PE searches the data structure, *join-list*, from the PE's local memory to determine which tokens in its memory should be joined with the received token. If the result of computing consistent variable bindings in *join-list* is successful, a new token is formed and sent to the appropriate bus. If at any instance, a token is sent to level 0, the match phase is terminated and the conflict resolution phase is started.

One interesting feature of this algorithm is that sharing in the Rete network is not lost as a result of distributively constructing the network. Moreover, no partitioning algorithm was proposed to partition nodes among PEs of a particular level. The estimated performance of the proposed algorithm on PESA-1 is 20,000 production firings

per second. It may also be a good idea to investigate the application of node-level and intra-node level parallelism using such an architecture.

2.3.6 Parallel Firing Mechanism

Unlike previous approaches attempted to improve performance of the match phase, the parallel firing mechanism [18] proposes a new parallel execution model where multiple productions are to be fired concurrently. This model is assumed to be implemented on a multiprocessor system that consists of a control processor (CP) and a large number of processing elements (PEs), where each PE has its own local memory. Although this mechanism was proposed to be implemented on DADO, it can be applied to both shared and distributed memory architectures since no particular inter-process communication is assumed. The processing performed in the Recognize-Act cycle is modified in order to support this mechanism as follows:

Match Phase: Productions are to be distributed between CP and PEs where productions having I/O operations are allocated to the CP. Each of the CP and PEs are to perform the match productions in parallel, where productions matched by the PEs are to broadcast to the CP.

Conflict-Resolution Phase: The CP selects one or more of the matched productions. The selection is no longer dependent on conflict resolution strategies, LEA and MEA. Alternatively, productions are selected based on their interference, where rule A is said to interfere with rule B if firing both produces contradictory results.

Act Phase: Each PE executes concurrently the actions of the production assigned to it by CP. Resulting WM changes are supposed to be accessed by all PEs regardless of whether the system is being executed on a shared or a distributed memory multiprocessor system.

Ishida and Stolfo [18] noted two problems with respect to implementing this model: synchronisation of productions firings and decomposition (or partitioning) of productions.

Synchronisation Problem

Synchronisation of production firing under the parallel firing mechanism means avoiding interference between any two productions in a production system cycle. To help detect such an interference between productions, a data dependency graph is to be constructed for all productions, where two types of nodes exist: production nodes P-node and WM-nodes, that are connected by directed edges. A directed edge from a P-node to a WM-node indicates that the RHS of the production in P-node adds or deletes a certain class of WM elements and is labelled $+/-$, in consequence. On the other hand, a directed edge from a WM-node to a P-node indicates that the LHS of a P-node references WM class of WM-node and is labelled $+/-$ for positive or negative condition elements, respectively. This type of graph can be used to produce synchronisation sets for every production if there exists a WM-class that satisfies any of the following criteria:

1. A WM-node of a class indicates addition or deletion by one production and negative or positive reference by some other rule(s).

2. A WM-node of a class indicates addition or deletion by one rule and deletion or addition by some others.

Decomposition Problem

Ishida and Stolfo proposed a decomposition algorithm to partition productions among processors. This algorithm has two phases. In the first, a tree of productions is to be constructed based on two heuristics: descending order of tokens and linked order of tokens[9]. In the second phase, partitions are created by selecting a suitable layer of the production tree.

Simulation of the parallel firing mechanism using 32 processors using the Manhattan MAPPER [25] production system showed a speed-up of 7.5 over Rete-based OPS5 on a uniprocessor system.

[9]Tokens have the following form in this approach: (prod1, prod2, p(prod1,prod2)), where prod1, prod2 are any two productions and p(prod1, prod2) is the parallel executability of production prod1 and prod2 defined as the number of cycles saved if prod1 and prod2 are placed on two distinct processors. The value of p is to be obtained from production system run traces.

2.4 Conclusion:

The result of this survey can be classified into research on uniprocessor and on shared or distributed memory multiprocessor systems.

Research in the uniprocessor environment may be summarised as either investigating better algorithms (e.g. Rete, TREAT) or exploiting new architectures (e.g. RISCF machine). While Rete is the most well-known, efficient and tested uniprocessor algorithm, it lacks the ability to match newly-created productions with the current configuration of working memory. Although TREAT can remove productions instantiations from the conflict-set whose positive condition elements were matched by WM elements that have already been removed. it cannot remove those instantiations whose negative condition elements have been matched by a newly inserted WM elements without recomputations of consistent variables bindings. Thus, TREAT partially utilised the conflict-set support conjectured in McDermott et. al [30]. In addition, Rete and TREAT rely heavily on state held in global variables which can have an impact on the degree of potential parallelism.

Research in the multiprocessor environment may be described as exploiting parallelism in a relatively sequential algorithm (i.e. Rete) by proposing parallel versions of it to run on special-purpose parallel hardware (e.g. DADO and PESA-1). The exception is the NON-VON machine which was used for parallel production system research but was designed as a general-purpose parallel symbolic architecture. The following are the main observations on these efforts:

- Only one of these algorithms has been implemented[10], hence all rest of the performance figures are from simulations. The exception is the full distribution version of TREAT on the DADO machine. However, this suffers some serious drawbacks in that it tries to exploit production level parallelism, which is now known to be of limited value, by allocating a PE for each production or set of productions, but the amount of memory per node is insufficient to hold the WM elements to be matched by these productions.

[10] as far as the author is aware

- Although there has been some attempts at partitioning the production memory across PM-level PEs to fit the DADO architecture this remains a problem for which an optimal solution is yet to be found.

- The highly parallel algorithm of Oflazer has two serious problems: (i) the redundancy check of instance elements has to be done serially; (ii) saving state in the instance elements requires vast amount of resources which led Oflazer to propose his parallel architecture.

- The parallel firing mechanism is concerned with the multiple firing of rules (i.e. fire phase of the Recognise-Act cycle), whereas the bottleneck in the execution of production systems lies in the match phase. In addition, there is no real implementation of this mechanism to validate its performance and correctness compared to the traditional OPS-like programs.

- Although many schemes have been proposed for parallel architectures for production systems, it seems unlikely that such machines could ever be commercially viable given their limited applicability. Thus, it is more practical and realistic to think of algorithms that utilise the available parallel hardware.

In conclusion, a case may be formulated that there is a need for a software architecture that is parallelizable, extensible and does not need a special purpose designed parallel architecture to execute production system programs as will be discussed in the succeeding chapters.

Chapter 3

Object-Oriented Execution of OPS5 Production Systems

The primary goal of this research is to improve the performance of production systems. But, unlike the previous attempts summarised in the last chapter, the main aspect here is that instead of starting from the existing program, namely the OPS5 interpreter, we began from a specification of that program derived from an informal analysis of the semantics of OPS5 rule sets and applied object-oriented software development techniques to define classes and methods on them to synthesize a new object-oriented OPS5 interpreter.

In this chapter, section 3.1 presents a brief background to object-oriented programming (OOP), section 3.2 demonstrates a uniprocessor object-oriented model for executing OPS5 production system programs. Section 3.3 is a discussion of the various aspects of this approach and is followed by a conclusion in section 3.4.

3.1 Object-Oriented Programming Background

Object-oriented programming can be traced back to SIMULA [6] when the concept of the object was first introduced as a program that has its own data and actions. Later, the concept of object was refined in SMALLTALK [13] to be an entity that has its own private memory (i.e. attributes and their values) and a set of operations defined

on it. Hence, an object-oriented system reduces to a set of objects that communicate together by exchanging messages. Moreover, each object in an object-oriented system is an instance of some class, where a class can be defined as an entity that has its own private memory, interface and methods or procedures. Thus, an object which is an instance of some class, shares the interface and methods parts of that class but has its own private memory which may be copied (e.g partially or fully) or assigned to when it is created.

Object-oriented languages are based on two fundamental concepts, *message sending* and *specialisation* [42]. *Message sending* is not only the means by which objects communicate but supports an important principle, namely data abstraction. This principle provides the control by which state and methods may be accessed. On the other hand, *specialisation* or hierarchy refers to the use of class inheritance to ease the creation of objects and structuring them in a hierarchy. Hence, this enables us to create subclasses that inherit some characteristics and behaviour (methods) of some superclasses.

3.2 Object-Oriented Execution of Productions Systems

An initial reaction might be "why use objects?". The answer to that is simple: object-oriented systems have a number of features that motivated the approach of object-oriented execution of production systems. These features are :

First, object-oriented systems are dynamically configured which when exploited relieves the execution process from being restricted by a fixed path as in the case of the Rete network [8]. As a result, this would make object-oriented systems more responsive to changes in their problem domains, which may be contrasted to production systems, where changes to WM are a continuous process during the execution of a production system.

Second, objects offer greater degree of encapsulation which has the following two implications :

- a correspondence between the production system representation of knowledge and the object notion when considering that each production has its own condition

elements and actions that are independent from the condition elements and actions of other productions.

- a higher degree of potential parallelism. The original OPS5 program relies very heavily on state held in global variables and, indeed, on the use of Lisp's `eval` function to access data stored by name. This should also reveal why we preferred to start from the idea rather than the program.

Thirdly, objects in object-oriented systems closely match our perception of our environment (i.e. how humans, things, etc. communicate) in which it leads to a better understanding of a problem domain as will be shown later when identifying the objects and their relationships to construct an object-oriented model of production systems.

Fourthly, object-oriented systems represent one of the platforms for parallel processing especially when considering the research being carried out in the field of concurrent OOP. Hence, transforming a production system into an object-oriented one enables much of the ideas explored in concurrent OOP to be explored in production systems execution.

However, studying the execution of production systems, it is obvious that the key initiators of a production system cycle (i.e. Recognize-Act cycle) are changes to WM which imply adding, deleting or modifying a WM element. Modifying a WM element is nothing more than the spawning of the two actions *remove* and *make*. A *remove* action removes a particular WM element, while a *make* action puts back the removed WM element along with the changes specified by the original *modify* action. As a result, one can conclude that it is *make* and *remove* messages that initiate a production system cycle if a production system is to be executed in an object-oriented environment.

The object-oriented execution of OPS5 production systems proposed and implemented in this research is a two phase process as shown in Figure 3-1. First, in order for a production system program to be executed in an object-oriented environment, it has to be transformed into a system comprised of a set of objects. This transformation is a kind of compilation which corresponds to building the Rete network in the traditional implementation. Thus in effect an *object-oriented compiler* (or an object-

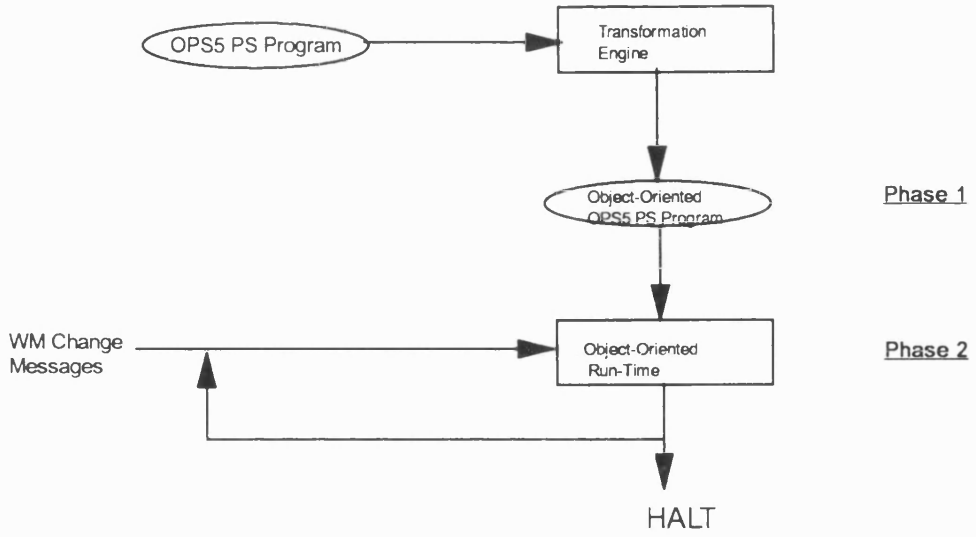

Figure 3-1: Object-Oriented Transformation and Execution of OPS5 Production Systems

oriented transformation engine) for OPS5 rules has been built and so it should occasion no surprise to learn that the semantics of OPS5 are preserved. Coupled to this is an *object-oriented run-time* system to handle WM-Change messages and initiate the execution of the transformed object-oriented system. The transformation engine has been developed using EuLisp[34] and, to be more precise, the EuLisp object system known as TEAOΣ[2].

Prior to the design process, a well-known object-oriented development methodology, the one described by Booch [1], was used to design the new implementation of OPS5, which we have named OOPS5. The five stages Booch identifies for the design process are:

1. Identification of the objects and their attributes;
2. Identification of the behaviour of each object;
3. Establish visibility of each object;
4. Establish interface of each object;
5. Implement each object.

```
(literalize c1 c11 c12)

(literalize c2 c21 c22)

(literalize c3 c31 c32)

(p p1
   (c1 ^c11 2 ^c22 <x>)
 - (c2 ^c21 <y> ^c22 <x>)
 - (c3 ^c31 50 ^c32 100)
  -->
   (make c2 ^c21 3 ^c22 5)
   (make c3 ^c31 50 ^c32 100)
   (remove 1))

(p p2
   (c1 ^c11 2 ^c22 <x>)
 - (c2 ^c21 <y> ^c22 <x>)
  -->
   (write p2 is successful))

(p p3
   (c1 ^c11 2 ^c22 <x>)
 - (c3 ^c31 50 ^c32 100)
  -->
   (write p3 is successful))

(p p4
   (c1 ^c11 2 ^c22 <x>)
  -->
   (write p4 is successful))
```

Figure 3-2: An example of an OPS5 rule-set

Note that for the purposes of this discussion, it has been chosen to be consistent with the terminology of Booch in referring to *objects* and *attributes*, although these are probably more widely referred to as *classes* and *slots*, respectively. Similarly, the *suffered* methods refer to methods that may be invoked by other objects and *required* methods to methods invoked by an object on itself, rather than the public and private nomenclature of C++ for example.

We present the application of Booch's methodology in the order of the steps he defines with the addition of examples to illustrate the first and second steps.

3.2.1 Identification of objects

This step is usually preceded by a requirements analysis phase to clarify the problem domain. However, understanding the theory of production systems and their execution is assumed to suffice. The example of Figure 3-2 is used to show how a given rule-set (a small production system program written by the author to show the key features of OOPS5) is transformed into respective objects. The following are the six objects used by OOPS5: Productions, Condition-Elements Objects (CE-Objects), WM-Distributors, Conflict-Resolution (CR) Manager, Working Memory (WM) and WM-Clock.

Productions

Each production rule of the original program is considered as an object that serves two purposes. First, it controls the computation of consistent variable bindings between condition elements of the same production and hence contributes indirectly to the insertion of its instantiation into the conflict-set. Second, it is responsible for firing the actions in its right hand side.

This object has four attributes. The way in which a production is used depends heavily on the form of the left hand side conditions. It is convenient to have one attribute to indicate whether the condition elements are all positive, negative, or a mixture. This is called the `classification` attribute.

The `match knowledge` of a production is information associated with each condition element consisting of (i) a unique identifier naming the condition element, which is constructed from the production name, the class of the condition element and its sequence number within the production (ii) a flag to indicate whether the condition element is *joinable* or *nonjoinable* (the term *joinable* is used to denote whether a condition element computes consistent variable bindings with one or more condition elements of the same production whereas the term *nonjoinable* refers to the opposite) (iii) whether this particular element is positive or negative (iv) a count of the number of WM elements matched by this condition element in processing a new WM change.

The right hand side of a production is the `actions` attribute. The last attribute is the `rating`, which is a count of the constant and variable tests in all the condition

elements of the production.

The following are the production objects of the rule-set in Figure 3-2:

1. Production p1 has the following attributes:

```
match-knowledge:        (p1-c1-1 J P  0)
                        (p1-c2-2 J N  0)
                        (p1-c3-3 N N  0)
classification: mixed
actions:        (make c2 ^c21 3 ^c22 5)
                (make c3 ^c31 50 ^c32 100))
                (remove 1))
```

   rating: 6—obtained by counting the number of tests in the left hand side.

2. Production p2 has the following attributes:

```
match-knowledge:        (p2-c1-1 J P 0)
                        (p2-c2-2 J N 0)
classification: mixed

actions: (write p2 is successful)

rating: 4
```

3. Production p3 has the following attributes:

```
match-knowledge:        (p3-c1-1 N P 0)
                        (p3-c3-2 N N 0)
classification: mixed

actions: (write p3 is successful)

rating: 4
```

4. Production p4 has the following attributes:

```
match-knowledge: (p4-c1-1 N P 0)

classification: positive

actions: (write p4 is successful)

rating: 2
```

Condition Elements

The condition elements of a production are represented as objects in this formulation rather than as paths through a network as in the Rete algorithm. It is this change of representation that makes incremental addition of rules possible at run-time. These objects are also referred to in this work as CE-objects. The role of a CE-Object is to store WM elements satisfying its constant and variable tests and to hold some control information to be utilised when joining it with other CE-objects. A unique name is constructed for each CE-object as described above. This name represents an instance of the CE-object class. Each CE-object has the following six attributes described below.

A `type` attribute is used to record whether this particular condition is either positive or negative. The `CE-tests-list` attribute is a list of tuples corresponding to the constant and variable tests in a condition element. Each tuple comprises a `type`, a `predicate` and a `value`. In a condition the type is taken to be a constant test (c), a join variable (jv), or a free variable (fv), that is a non-join variable that is used either for binding on the right hand side, or as a dummy variable. The predicate is one of the defined OPS5 predicates. The value field gives the actual constant or variable name, as specified by the type field.

Whether a condition element has join variables or not is an important feature of a condition element, and so this is recorded in the `join` attribute.

WM elements matching a condition element are stored as tuples in an AVL tree [21], where their keys are time stamps generated by the WM-Clock object (see later). This attribute is called `WM-AVL-tree`. The selection of AVL trees to store matching WM elements is based on two important points. First, AVL trees are good at information retrieval if few insertions or deletions are performed. Empirical measurements reported in [15] on the rate of change of working memory — that is the number of additions and deletions to working memory — is between 2 and 4 actions per recognize-act cycle, which is sufficiently small to support the choice of AVL trees. Second, some OPS5 production system programs require initial loading of very high number of working memory elements — for example. Mapper [25] requires an initial configuration of 576 WM elements — and hence using AVL trees would be beneficial at retrieval time.

An additional AVL tree is used for each join variable in a particular condition element to store the values of join variables associated with working memory elements matching that condition element. The `CE-AVL-trees` attribute associates the names of these trees with their corresponding join variables.

A CE-Object is said to be unique if there does not exist any other CE-Object that has the same `CE-tests-list`, same `type` and same joinability. The significance of this uniqueness is that it avoids creating redundant CE-Objects. The name or names of the production(s), of which the condition element is part of is recorded in the object as the `productions` attribute. As a result, `p1-c1-1` and `p1-c2-2` replace `p2-c1-1` and `p2-c2-2`, respectively, in the match-knowledge of production `p2`. Also, `p1-c3-3` replaces `p3-c3-2` in the match-knowledge of production `p3` whereas `p3-c1-1` replaces `p4-c1-1` in the match-knowledge of production `p4`.

Following the uniqueness concept introduced above, the CE-Objects of the rule-set in Figure 3-2 are:

1. `p1-c1-1`

   `productions: p1 p2`

   `type: positive`

   `join: join`

   `WM-AVL-tree: ()` (initially)

   `CE-AVL-trees: (x equal p1-c1-1x)`

   `CE-tests-list: (1 . (c equal 2)) (2 . (jv equal <x>))`

2. `p1-c2-2`

   `productions: p1 p2`

   `type: negative`

   `join: join`

   `WM-AVL-tree: ()` (initially)

   `CE-AVL-trees: (x equal p1-c2-2x)`

CE-tests-list: (1 . (fv equal <y>)) (2 . (jv equal <x>))

3. p1-c3-3

   productions: p1 p3

   type: negative

   join: non-join

   WM-AVL-tree: () (initially)

   CE-AVL-trees: ()

   CE-tests-list: (1 . (c equal 50)) (2 . (c equal 100))

4. p3-c1-1

   productions: p3 p4

   type: positive

   join: non-join

   WM-AVL-tree: () (initially)

   CE-AVL-trees: ()

   CE-tests-list: (1 . (c equal 2)) (2 . (fv equal <x>))

WM distributor

Instances of the `WM-Distributor` class serve as distributors of working memory change messages to CE-Objects instances of a particular entity. Instances of the WM-Distributor class have names that correspond to the objects defined in `literalize` statements in the original OPS5 program. A WM-Distributor object has one attribute, namely the `CE-Objects-list`. This attribute maintains a list of all the CE objects which belong to this particular instance. The WM-Distributors of the rule-set in 3-2 correspond to the entities defined in the OPS5 `literalize` statement of this rule set, which are `c1`, `c2` and `c3` where each is assigned `((p1-c1-1) (p3-c1-1))`, `(p1-c2-2)` and `(p1-c3-3)` as values of their CE-Objects-list attribute, respectively.

Conflict-Resolution Manager

This object is responsible for maintaining the conflict set and computing the conflict-resolution. One attribute is the `resolution-strategy` — either LEX or MEA[1]. The other is `conflict-set`, which is a list of instantiations of productions resulting from the match phase of the recognize-act cycle being tuples of (i) a production instance, (ii) variable bindings, (iii) timestamps associated with each working memory element matching a positive CE-object and (iv) the rating of the production. It is assumed that there is one conflict-resolution manager, called `Manager1`.

Working Memory (WM)

The WM object is used to store all the working memory elements ever inserted. This object has one attribute, namely `WM-vector`, which is treated as a two-dimensional array. Access uses the timestamp of WM element as the first key and the indices of its attributes as the second key. There is only one instance of WM, namely `WM1`.

WM-Clock

When a WM element is to be inserted into WM it has to have a timestamp to indicate the time at which it was inserted which has implications for the overall execution process and specifically on the computation of conflict-resolution and movements of production instantiations into and out of the conflict set. Hence, there is a need for a clock that keeps and advances WM time. This implies using it to generate timestamps for each WM element to be stored in the `WM-vector` attribute of the WM object. This clock object has one attribute, `WM-Clock-time`, which is an integer representing WM-Clock's time at any instant during execution. The only instance of `WM-Clock` is `WM-Clock1`.

3.2.2 Identify behaviour part of each object

The behaviours of objects can be further segregated into static, being the methods defined on an object, and dynamic, being time or space constraints on an object [1]. In

[1] See [10] for details of these conflict-resolution strategies.

this exposition we describe these two aspects together. For ease of presentation in this work, a message sent to an *object* is assumed to have the following form:

$\Longrightarrow$ **Target-Object Method-Name:** $parm_1$, $parm_2$, ... ,$parm_n$

To help present the underlying algorithm that controls the execution of a derived object-oriented production system program, the behaviour of each of the objects introduced earlier is given here in the order of execution, where possible. In the preceding section, it was preferred to use a top-down order of presentation hoping that giving both offers a more intuitive understanding of the structure of the system to the reader in the differing circumstances.

WM-Clock

There is only one method that characterises the behaviour of this object, namely `retrieve-WM-time`. This is a suffered (that is externally called) method on the WM-Clock object which returns the result of advancing WM-clock-time by one unit. Sending a message to WM-Clock object requesting the retrieval of WM time is done both in the initial step of the overall execution process and prior to sending a message to WM object requesting an insertion of a WM element.

Working Memory Object

The behaviour of the working memory (WM) object is characterised by three methods, two of which are suffered and the third is a required one. The suffered methods are `insert-wme` and `retrieve-wme-entity` and the required one is `retrieve-wme`.

A request for the WM object to execute its `insert-wme` method is done at two places: either at the top level when loading the system with the initial WM elements or when a production object executes its `fire` method (see below). The processing accomplished by this method depends on the type of working memory action:

1. If it specifies insertion of a new WM element (argument), then insert it into `WM-vector` attribute of the WM object using a timestamp (argument) as an index, after resolving all variable bindings in this WM element using `free-list`

(argument). Note that a free-list is a property list that includes variable bindings of a production's instantiation.

2. Otherwise, if it specifies a modification to an existing working memory element in the `WM-vector` attribute, then a message is sent to WM object requesting the execution of the required method `retrieve-wme` which returns a WM element from the `WM-vector` using timestamp (argument) as an index. Then, a message is sent to WM-Clock object to get a new timestamp which is used to insert a new WM element which is constructed from the old one (the one just retrieved) and the modifications (argument). The free-list (argument) is used to resolve any variable binding in the modifications.

`Retrieve-wme-entity` is called by the WM object as a result of being called by the production object while executing the fire method and specifically when processing a modify action. This method returns the class[2] of a WM element in the `WM-vector` using a timestamp (argument) as an index. This method uses the `retrieve-wme` required method in retrieving this WM element.

WM-Distributor

WM-distributor has one suffered method, `broadcast-WM-change`, with two arguments. The first indicates the kind of working memory change (either addition or deletion) which is required, and the second is a timestamp. If the change is an addition, a `match-insert` message is sent to each CE-Object in the list maintained in the distributor. If the change is a deletion, a `remove-from-cs-timestamp` message is sent to the conflict resolution manager telling it to remove any production instantiations from the conflict set that have a WM element stamped with the given timestamp. In addition, a `remove` message is sent to each CE-object in the `CE-Objects-list`.

[2]The class is meant to be the one defined in the `literalize` statement

Condition Elements

Condition elements have three suffered methods, `match-insert`, `remove` and `join`, and have five required methods one of which is the `join` method. The other four methods are: `join-join`, `join-nonjoin`, `solve-positive-join` and `solve-negative-join`. The required methods are only discussed in passing here when they are used by the suffered methods.

The `match-insert` method: compares the results of computing the constant tests stored in the `CE-tests-list` attribute against the corresponding values in a WM element with a given timestamp (argument). The WM element will just have been inserted into the `WM-vector` of the WM object. If the tests are successful two property lists are constructed: (i) the *free-list* which associates variables (join or non-join) and their values (ii) the *join-list*, which associates a join-variable with a tuple of a predicate and a value. Then the balanced trees, `WM-AVL-tree` and those of `CE-AVL-trees`, of the CE-Object are updated as follows:

1. Construct a WM-tuple, which comprises the given time-stamp, free-list and join-list, and insert into WM-AVL-tree using the time-stamp as the key.

2. For each join variable, construct a list of timestamp, free-list and join-list and insert it into the AVL tree of this join variable in CE-AVL-Trees[3] attribute using the join variable's value as a key. However, if this key exists, the timestamp in the value is updated with the new timestamp.

Finally, the method sends `update-match-knowledge` to the production stored in the production attribute of the condition element. Details of the processing accomplished by the `update-match-knowledge` and its parameters will be described at the time of describing the behaviour of the production objects.

Consider for example that `p1-c1-1` CE-object[4] receives the following message :

$\Longrightarrow$ `p1-c1-1 match-insert:` T_1

Then, `match-insert` performs the following steps:

[3] Recall that each join variable has a corresponding AVL tree in CE-AVL-trees

[4] Obtained from the transformation of the rule-set in Figure 3-2

1. $\Longrightarrow$ `WM1 retrieve-wme:` T_1

   (assuming that the WM element (**make c1 ^c11 2 ^c12 5**) has already been inserted by **insert-wme** method and was given the timestamp T_1)

2. The evaluation of the constant tests stored in **p1-c1-1** returns true and hence the following two lists are constructed:

   free-list : ((x . 5))

   join-list : ((x . (equal . 5)))

3. WM-AVL-tree of **p1-c1-1** is updated with the tuple ((x . 5) (x . (equal . 5))) using T_1 (timestamp) as a key.

4. There is only one AVL tree in CE-AVL-trees of **p1-c1-1** associated with the join variable x namely, **p1-c1-1x1** which gets updated with the tuple: (T_1 ((x . 5)) ((x . (equal . 5)))) using 5 (value of **x**) as a key.

5. Update the match-knowledge of the corresponding productions in productions attribute of **p1-c1-1** by sending the following messages :

   $\Longrightarrow$ `p1 update-match-knowledge: +1, p1-c1-1,` T_1,

   (T_1 . p1-c1-1), ((x . 5)), ((x . (equal . 5)))

   $\Longrightarrow$ `p2 match-insert: +1 p1-c1-1,` T_1,

   (T_1 . p1-c1-1), ((x . 5)), ((x . (equal . 5)))

The remove method: This method gets called by a CE-Object as a result of receiving a remove message from a WM-Distributor to process the removal of a WM element of timestamp T (argument). Hence, this entails undoing the effect of matching this WM element when it was processed by a **match-insert** method. This involves undoing the updates on the balanced trees of the CE-Object and the effects on its respective production object(s) as a result of processing an **update-match-knowledge** method (see later). Undoing both of these effects is to be carried out only if a WM-tuple of timestamp T exists in the WM-AVL-tree of the CE-Object being operated on and is done as follows:

1. Remove WM-tuple stored in WM-AVL-tree of CE-Object using T as a key.

2. Use values of join variables in join-list of the removed WM-tuple to update corresponding CE-AVL-trees.

3. Send `update-match-knowledge` message to corresponding production object(s). Arguments to this message are: -1 (to indicate a removal of a WM element) and the name of the CE-Object being operated on.

For example, if `p1-c1-1` CE-Object receives a message to remove WM-tuple of timestamp T_1 that was inserted above in the example of the `match-insert` method, then the following steps are executed by the `remove` method:

1. WM-tuple `((x . 5) (x . (equal . 5)))` of timestamp T_1 is removed from WM-AVL-tree of `p1-c1-1`.

2. The only tree in CE-AVL-trees of `p1-c1-1` is the one associated with the join variable `x` which is bound to 5 in the join-list of the WM-tuple. Hence, tuples in this tree that a has a value of 5 (key) and a timestamp of T_1 (argument to `remove`) gets updated by removing the tuple: (T_1 `((x . 5)) ((x . (equal . 5))))` from this tree using 5 (value of x) as a key.

3. Update the match-knowledge of the corresponding productions in production attribute of `p1-c1-1` by sending the following messages :

   $\Longrightarrow$ `p1 update-match-knowledge: -1, p1-c1-1`

   $\Longrightarrow$ `p2 update-match-knowledge: -1, p1-c1-1`

`The join method`: The purpose of this method is to control joining WM tuples that matched CE-Objects of the same production (i.e. in Rete terminology, this means computing consistent variable bindings between different condition elements of a production). Initially, this message is sent by a production object while executing either `test-join-positive` or `test-join-mixed` methods in order to notify one of its CE-Objects to start the join process. Later this method becomes a required one in a sense that one CE-Object sends join messages to another one in the CE-Objects-join-list (argument). The parameters passed to this method are:

`production`: Name of production whose CE-Objects are to compute join.

`timestamps`: The accumulated list of timestamps of WM-tuples satisfying the join of the CE-Objects of a production.

`CE-Objects-timestamps`: A property list associating a CE-Object with a timestamp of a WM-tuple satisfying the current join in the CE-Object's WM-AVL-tree.

`free-list`: The incremental union of the free-lists related to WM-tuples in WM-AVL-trees of CE-Objects of a production satisfying the current join.

`join-list`: The incremental union of join-lists related to WM-tuples in WM-AVL-trees of CE-Objects of a production satisfying the current join.

`CE-Objects-join-list`: A list of the CE-Objects of a production—except the CE-Object affected by the initial WM change—ordered as positive joinable before negative joinable before positive nonjoinable.

The processing done by this method depends on whether the CE-Object is joinable or not as follows:

Processing of Non-Joinable CE-Objects

This task is handed off to the `join-nonjoin` required method. It computes a dummy join between WM-tuples in WM-AVL-tree of the CE-Object (CEO_1) being sent a `join-nonjoin`message and free-list (argument of join-nonjoin). The term "dummy join" introduced here means no computation of consistent variable bindings between CE-Objects as opposed to the classification of these CE-Objects as non-joinable ones. The result of such a join is the union of each of the free-lists in WM-tuples of (CEO_1) and free-list (argument) with the result of recursing on the rest of `CE-Objects-join-list`. When `CE-Objects-join-list` is empty, `join-nonjoin` sends a `insert-into-conflict-set` message to the conflict resolution manager. Details of processing in this method is shown in Figure 3-3. Note that parameters passed to this method are the same as the parameters passed to the `join` method.

1. L_0 ← CE-Objects-join-list
2. For each WM-tuple of timestamp T in WM-AVL-tree of CE-Object being operated on do;
   (a) Perform SET UNION operation on free-list of WM-tuple of T and *free-list* (argument to join-nonjoin) method. Name the resultant set F_1.
   (b) Construct a pair comprised of current CE-Object and T and then insert it into CE-Objects-timestamps (argument).
   (c) Insert T into timestamps (argument).
   (d) If L_0 is empty, construct a production instantiation of production object (argument) and send `insert-into-conflict-set` message to CR-Manager object and EXIT.
   (e) Otherwise, do the following :
      i. O_1 ← POP(L_0)
      ii. $\Longrightarrow O_1$ `join-nonjoin`: . *timestamps, CE-Objects-timestamps,* F_1, *join-list,* L_0
   (f) Enddo.

Figure 3-3: Details of Processing in `join-nonjoin method`

Processing of Joinable CE-Objects

The task of this method is to compute an incremental join between joinable CE-objects of the same production in order to complete the processing of joinable CE-objects in `CE-Objects-join-list` (argument), while processing of nonjoinable ones is passed off to the `join-nonjoin` method, if any. However, if there are no nonjoinable CE-objects, an `insert-into-conflict-set` message is sent to the CR-manager to insert an instantiation of the production (argument) using these CE-objects. Basically, this incremental join deals with the computation of consistent variable bindings between variable bindings in join-list (argument) and the join-lists of WM-tuples matching the CE-Object being operated on.

The basic strategy employed in carrying out this type of join is based on finding a join variable in `join-list` (argument) corresponding to join variables in a `CE-AVL-trees` attribute of a CE-object being operated on. Satisfying this condition is dependent on whether this CE-Object is positive or negative. If it is positive, then one join variable is assumed to suffice. Alternatively, if it is a negative one, then all join variables in CE-AVL-trees attribute should have corresponding ones in the join-list (argument).

The reason behind this is that the semantics of negative joinable CE-Objects require that no WM elements that match them have variable bindings that are consistent with positive CE-Objects of the same production. However, this condition is satisfied by default for negative CE-Objects because of the definition of join order introduced in this thesis, where positive joinable CE-Objects are joined first then negative joinable ones (if any) and then positive non-joinable ones (if any). Hence, by the time negative joinable CE-Objects are to be processed by this method all their join variables must have corresponding ones in join-list (argument).

But, what happens if a join variable of a positive CE-Object does not have a corresponding one in join-list (argument)? This implies that no join computations can be performed between join-list (argument) and any of the balanced trees of the CE-Object being operated on. Thus, it is necessary to search for a positive CE-Object in CE-Objects-join-list (argument) to find a one that has a corresponding join variable in join-list (argument). However, if there are no more positive CE-Objects to be joined or the search for such a join variable fails, then the current CE-Object being operated on (which has already failed to have a join variable) performs a dummy join between its join-lists and join-lists of WM-tuples stored in its WM-AVL-tree and the join-list (argument). This results in sending a number of `join` messages to to the next CE-Object in CE-Objects-join-list (argument) that are equivalent to the number of the WM-tuples. Alternatively, if a join variable is found, then the processing below takes place as if the CE-Object found to have a join variable is the current CE-Object being operated on.

The purpose of finding a join variable is to run a query on its associated AVL tree. The type of this query depends on the predicate associated with that join variable. There are six kinds of queries[5] used here corresponding to the six predicates defined in OPS5. The concept of query is used here to get a set of tuples of different timestamps.

However, if running such a query results in no tuples, then depending on whether the CE-Object being operated on is positive or negative, the processing is terminated or a join message is sent to the next CE-Object in CE-Objects-join-list (argument),

[5]Each query has a worst case performance of $O(\log n)$, where n is the number of tuples in the AVL tree.

1. L_0 — CE-Objects-join-list
2. For each tuple of timestamp T_i in tuples of query
   (a) Compute consistent variable bindings between join-list of T_i and join-list (argument). If that computation succeeds and returns a new join-list, then go to next step, otherwise EXIT.
   (b) Insert free-list of tuple T_i into free-list (argument).
   (c) Construct a pair comprised of current CE-Object and T_i and then insert it into CE-Objects-timestamps (argument).
   (d) Insert T_i into timestamps (argument).
   (e) If L_0 is empty, construct a production instantiation of production object (argument) and send **insert-into-conflict-set** message to CR-Manager object and QUIT.
   (f) Otherwise, do the following :
      i. O_1 — POP(L_0)
      ii. $\Longrightarrow O_1$ **join**: . *timestamps, CE-Objects-timestamps, free-list, new join-list,* L_0
   (g) Enddo.

Figure 3-4: Details of Processing in **solve-positive-join method**

respectively. Otherwise, each of these tuples has its own join-list (as it was stored while processing **match-insert** method) which it needs to be checked against consistent variable bindings of the join-list (argument). At this stage this method hands off this type of computation to either **solve-positive-join** or **solve-negative-join** for positive or negative CE-objects, respectively.

The computation handed off to the **solve-positive-join** method is a complete delegation of processing. In other words, the **join-join** method does not need any feedback from **solve-positive-join** method. Arguments passed to this method are the same as the ones passed to the **join-join** method but with an extra argument. This argument is the list of tuples resulting from running the query (as explained above), where each tuple is a list comprising *timestamp, join-list* and *free-list.* Details of the processing done in this method is described in Figure 3-4.

On the other hand, the computation handed off to the **solve-negative-join** method returns a logical answer to the **join-join** method as to whether there exists at least one tuple in WM-tuples (argument) that has join variable bindings in its join-list that

satisfy the join variable bindings in join-list (argument). If this answer is true, then processing in `join-join` method is terminated. Otherwise, a `join` message is formulated and sent to the next CE-Object in CE-Objects-join-list, if any.

Conflict-Resolution Manager

A Conflict-Resolution Manager (CR-Manager) object is considerably more complex, and suffers six methods.

The first, `insert-into-conflict-set` adds a production instantiation (argument) to the conflict-set. A production instantiation comprises the following elements: (i) production object's name, (ii) bindings of variables (join-list), (iii) a list of the timestamps associated with each WM element matching a positive condition element and (iv) the rating of the production object.

The method `remove-from-cs-timestamp` handles the removal of production instantiations from the conflict-set whose list of timestamps contain timestamp of a WM element removed that was matched by a positive CE-Object. In this way and similar to TREAT algorithm, no join recomputations are needed as a result of this removal because previous join computations were not saved as in the case of beta memories in the Rete algorithm. This method is called after finishing the processing related to removing of a WM element and specifically after the end of processing of `remove` messages in the `broadcast-wm-change` method.

The method `remove-from-cs-production` is called after finishing matching of a newly inserted WM element that matches a negative nonjoinable CE-object of a mixed production, P_x. This method removes production instantiations of P_x and hence this avoids any join recomputations as in the cases of both Rete and TREAT algorithms.

The method `check-removal-of-production` deals with the effect of adding a new WM element that matches a negative joinable CE-object by removing any production instantiations of a production (argument) that have bindings of join variables that satisfy the constraints on the bindings of the same variables in a given join-list (argument). For the definition of join-list see `match-insert`.

In summary, the two methods `remove-from-cs-production` and `check-removal-`

`of-production` make a major algorithmic shift from TREAT and Rete in that the methods avoid join recomputations as a result of inserting a WM element that matches a negative condition element.

Computing the conflict resolution according to the strategy attribute of the CR-manager is done by `compute-conflict-resolution`; the strategy is either LEX or MEA following [8].

It is possible to switch the strategy attribute from LEX to MEA or vice versa, by using the `change-strategy` method. Note however that this operation should only be done at top-level and before processing `compute-conflict-resolution`.

Productions

A Production object suffers two methods, `update-match-knowledge` and `fire`. It also has three required methods, one of which is the suffered `fire` method. The other two are `test-join-positive` and `test-join-mixed`. The following is a description of the processing performed by each of these methods:

The main purpose of the `update-match-knowledge` method is to update the match knowledge of a corresponding CE-Object. The CE-object is an argument passed to this method that has just been processed by either `match-insert` or `remove` messages. Details of the processing in this method and reasons behind decisions at critical points are given in Figure 3-5. The method is also passed a tag to indicate whether the WM-tuple has just been added $(+1)$ to or removed (-1) from the WM-AVL-tree of a `CE-Object` and the timestamp of the WM-tuple that has just been inserted. In addition, three property lists are provided to the method, the first, `CE-Objects-timestamps`, associating the `CE-Object` and `timestamp`, and the second, `free-list`, consisting of elements that were initially extracted from a WM-tuple matching `CE-Object`, of free variables and their values. The last property list, `join-list` associates the join variables with the predicate and value, extracted from WM-tuple of `timestamp`.

The required methods `test-join-positive` and `test-join-mixed` control the joining of CE-Objects as they decide not only whether to carry out this join but in which order it should be carried out. These methods utilise knowledge available in the match-

1. **previous-count** ← **this-count** (The value of **this-count** is extracted from the match-knowledge of the *CE-Object* (argument).
2. Increment/Decrement **this-count** value in the match-knowledge of *CE-Object* denoting insertion and deletion of WM-tuple, respectively. Hence, match-knowledge of *CE-Object* is updated.
3. If *CE-Object* is negative, then do the following :
   (a) If *indicator* = +1, then if *CE-Object* is joinable, then send **test-join-mixed** to production itself to test its eligibility to compute join and then send **check-removal-of-production** to Manager1. Otherwise, (i.e. if CE-Object is non-joinable), do the following:
      i. If **previous-count** = 0, which implies the following: (1) There were no WM-tuples in WM-AVL-tree of *CE-Object* before insertion of WM-tuple of timestamp, T. (2) There exists currently WM-tuple in WM-AVL-tree of *CE-Object* as a result of having WM-tuple of timestamp T been inserted into WM-AVL-Tree of *CE-Object*. Then send **remove-from-cs-production** message to the CR-Manager, to remove any productions instantiations having current production object.
      ii. If **previous-count** > 0, which implies the following: (1) There exists currently WM-tuples in *CE-Object's* WM-AVL-Tree. (2) There existed WM-tuple(s) in *CE-Object's* WM-AVL-Tree as a result of previous WM-tuple(s) inserted. As a result, processing is terminated.
   (b) Else if (i.e. indicator=-1), then send **test-join-mixed** message to production object itself to test its eligibility to compute join as a result of this removal.
4. Else if *CE-Object* is positive, then do the following
   (a) If *indicator* = *-1*, then terminate processing of update-match-knowledge because WM-tuple of timestamp T has already been removed from WM-AVL-Tree of *CE-Object* and any no further invocations of **join** method are needed for the following reasons: (1) A **remove-from-cs-timestamp** message will be sent later (see **broadcast-wm-change** method) to **Manager1** (CR-Manager) to remove all productions instantiations having timestamp T, (2) In this approach, as in TREAT, no state is saved with respect to previous computation of consistent variable binding between CE-Objects as is done in Rete.
   (b) Else, if **this-count** > 0, then depending on classification of production object whose match-knowledge is being processed, send **test-join-positive** message to this production object if it is positive or **test-join-mixed** message if it is mixed or negative.

Figure 3-5: Details of Processing in **update-match-knowledge**

knowledge of a production as well as some heuristics.

Test-Join-Positive

The purpose of this method is to decide whether to carry out join between CE-Objects of a positive production and in which order to do it. Parameters passed to this method are the same as passed to `update-match-knowledge` except the `tag` parameter is normally omitted. The steps carried out by this method are:

1. If any CE-Object has a value of 0 for its `this-count` field in the match-knowledge of a production, then processing in this method is terminated. This is because the CE-Object (positive) is not matched by any WM element.

2. Otherwise, construct `CE-Objects-join-list` which is a list of all CE-Objects of the production being operated on except the CE-Object (argument). The joinable CE-Object precede the nonjoinable one in this list. Then, a `join` message is then sent to the first CE-object in CE-Objects-join-list.

Test-Join-Mixed

This method decides whether a join should be carried out between CE-Objects of a mixed or negative production. Parameters passed to this method are the same as for `update-match-knowledge`. It works as:

1. If at least one nonjoinable and negative CE-Object has a `this-count` value greater than zero, then finish processing.

2. If at least one positive CE-Object has a `this-count` value of zero, then finish processing.

3. Otherwise, do the following:

   (a) If CE-Object (argument) is negative and joinable and `tag` indicates an addition of WM element, then send a `check-removal-of-production` message to `Manager1` (CR-Manager).

(b) Otherwise, construct `CE-Objects-join-list` in which the CE-Objects are ordered with positive joinable before negative joinable before positive non-joinable except for the CE-Object (argument). Then, send a `join` message to the first CE-Object.

Unlike Rete which assumes a fixed order path in the Rete network, the join computation here does not assume any predefined order but a dynamic ordering at execution time[6]. In carrying out join between CE-Objects of the same production either of two strategies were employed: (1) positive joinable CE-Objects are joined to negative joinable ones, if any, and then to positive non-joinable if any (2) same as the first strategy except that CE-Objects are ordered in increasing size of their WM-AVL-trees within each group of joinable CE-Objects (i.e. positive joinable or negative joinable). The performance of these two strategies is compared in the next chapter.

The last method is `fire`, which is both suffered and required. However, it is suffered only once, prior to executing the first production system cycle and required thereafter. The `fire` method corresponds to the act phase in the Recognize-Act cycle but the way actions are executed differs. Parameters passed to this method are `free-list` and `CE-Objects-timestamps`, which are as described for the parameters of `join`. Figure 3-6 is an algorithmic description of the processing of this method.

3.2.3 An Example of Object-Oriented Execution

This section demonstrates the interaction between the objects obtained from the rule-set in Figure 3-2 when the following action is used to initiate execution:

```
(make c1 ^c11 2 ^c12 5)
```

The following messages are to be sent to initiate execution:

1. $\Longrightarrow$ `WM-Clock retrieve-wm-time`

   This results in retrieving a timestamp, call it T_1.

[6]Section 3.3 discusses the reasons behind this ordering.

1. For each `action` in `actions` attribute DO;
   (a) If `action` is MAKE, then do the following:
      i. Bind variables in MAKE, if any, to their corresponding values in *free-List* and then send `retrieve-wme-timestamp` message to WM-Clock1 to get timestamp T of WM element to be newly added.
      ii. Send `insert-wme` message to `WM1` object to insert new WM element in `WM-vector` using timestamp T as an index.
      iii. Send `broadcast-WM-change` message to corresponding WM-Distributor object whose name is the entity name of the WM element in the MAKE action.
   (b) Else. if `action` is MODIFY, then do the following:
      i. Use *free-List* to bind variables in MODIFY if any.
      ii. Use *CE-Objects-timestamps* to get corresponding timestamp, T_1 of WM-tuple that is to be modified and then send `retrieve-wme-entity` message to `WM1` object to get entity of that WM-tuple using T_1 timestamp.
      iii. Send a `broadcast-WM-change` message to corresponding WM-Distributor object, whose name is the entity of the WM element being modified, asking it to remove WM-tuple of T_1 timestamp.
      iv. Send an `insert-wme` message to `WM1` object to insert a new WM element that is composed of WM-tuple of T_1 timestamp and the new modifications included in the MODIFY action.
   (c) Else if `action` is REMOVE, then do the following :
      i. Use *CE-Objects-TimeStamps* to get corresponding timestamp, T_1 of WM-tuple that is to be removed and then send `retrieve-wme-entity` message to `WM1` object to get class name of that WM-tuple using T_1 timestamp.
      ii. Send a `broadcast-WM-change` message to corresponding WM-Distributor object, whose name is the entity of the WM element to be removed, asking it to remove WM-tuple of T_1 timestamp and all its side-effects.
   (d) Else if `action` is HALT, then execution is terminated.
   (e) Otherwise, perform the function required as either an I/O function or others using *free-List* to bind values to parameters in functions, if any.
   (f) End DO
2. Send `compute-conflict-resolution` message to `Manager1`, the local conflict resolution manager object.
3. If there is a production instantiation to be fired, then the `fire` method is called to execute this production instantiation. Otherwise, execution is terminated.

Figure 3-6: Algorithmic Description of the fire Method

2. $\Longrightarrow$ `WM1 insert-wme:` T_1, `(make c1 ^c11 2 ^c12 5)`

   This results in inserting this WM element into WM-vector at index T_1.

3. $\Longrightarrow$ `c1 broadcast-WM-change:` T_1, `make`

At this stage, no more messages are to be sent by the user to initiate the execution unless another WM element is to be considered to initialise the execution. As a result of this message, `c1` WM-Distributor responds by sending `match-insert` messages to `p1-c1-1` and `p3-c1-1` CE-Objects as follows:

1. $\Longrightarrow$ `p1-c1-1 match-insert:` T_1

   The processing of this message was done while discussing the `match-insert` method and resulted in sending `update-match-knowledge` messages to `p1` and `p2` productions as follows:

   (a) $\Longrightarrow$ `p1 update-match-knowledge:` `+1`[7], `p1-c1-1`[8], T_1, (T_1 . `p1-c1-1`)[9], ((`x` . `5`))[10], ((`x` . (`equal` . `5`)))[11]

   As a result, `p1's` match-knowledge is updated with respect to `p1-c1-1` and because `p1` is a mixed production, it sends `test-join-mixed` message to `p1`. Since, `p1-c2-2` and `p1-c3-3` CE-Objects in the match-knowledge of `p1` are negative and not matched by any WM-tuples, then `p1` is eligible to compute join. But, since `p1` has one positive CE-Object, then this leads to send `insert-into-conflictset`to `Manager1` to insert the following instantiation of `p1` into conflict-set:

   $\Longrightarrow$ `Manager1 insert-into-conflict-set:`

   `p1, ((x . 5)),` (T_1 . `p1-c1-1`), `6`

   (b) $\Longrightarrow$ `p2 update-match-knowledge: +1, p1-c1-1,` T_1,

   (T_1 . `p1-c1-1`), ((`x` . `5`)), ((`x` . (`equal` . `5`)))

[7] indicator for insertion of a WM element
[8] CE-Object used in the insertion
[9] CE-Objects-timestamps
[10] free-list
[11] join-list

In consequence, p2 responds by updating match-knowledge of p1-c1-1 and because p2 is a mixed production, it sends itself the **test-join-mixed** message. Since, p1-c2-2 CE-Object in the match-knowledge of p2 is negative and not matched by any WM-tuples, then p2 is eligible to compute join. Noting that p2 has one positive CE-Object, this leads to send **Manager1** an **insert-into-conflict-set** message to insert the following instantiation of p2 into conflict-set:

$\Longrightarrow$ **Manager1 insert-into-conflict-set:**

p2, ((x . 5)), (T_1 . p1-c1-1), 4

2. $\Longrightarrow$ p3-c1-1 match-insert: T_1

The p3-c1-1 CE-Object responds by evaluating the constant tests stored in CE-tests-list which return true. Since this CE-Object is nonjoinable, then **WM-AVL-tree** is updated with the tuple ((x .5)) where ((x . 5)) is the free-list[12]. By the same reasoning, no AVL trees exist in CE-AVL-trees attribute and hence no more AVL trees are to be updated. Finally, **update-match-knowledge** messages are sent to p3 and p4 productions as follows:

(a) $\Longrightarrow$ p3 **update-match-knowledge:**

+1, p3-c1-1, T_1, (T_1 . p3-c1-1), ((x . 5)), ()[13]

As a result, p3's match-knowledge is updated with respect to p3-c1-1 and because p3 is a mixed production, it sends a **test-join-mixed** message to itself. Since, p1-c3-3 CE-Object in the match-knowledge of p3 is negative and not matched by any WM-tuples, then p3 is eligible to compute join. Since p3 has one positive CE-Object, then **insert-into-conflict-set** message is sent to **Manager1** to insert the following instantiation of p3 into conflict-set:

$\Longrightarrow$ **Manager1 insert-into-conflict-set:**

p3, ((x . 5)), (T_1 . p3-c1-1), 4

[12]There is no need to construct a join-list since p3-c1-1 is nonjoinable

[13]join-list is () because p3-c1-1 is nonjoinable

(b) $\Longrightarrow$ p4 update-match-knowledge:

+1, p3-c1-1, T_1, (T_1 . p3-c1-1), ((x . 5)), ()

Production p4 responds by updating the match-knowledge of p3-c1-1 and because p4 is a positive production, it sends itself **test-join-positive** message. Since, p3-c1-1 is the only CE-Object in p4 and positive, then p4 is eligible to compute join. This leads to send **insert-into-conflict-set** message to **Manager1** to insert the following instantiation of p4 into conflict-set:

$\Longrightarrow$ **Manager1 insert-into-conflict-set:**

p4, ((x . 5)), (T_1 . p3-c1-1), 2

Now, since no more WM actions are to be executed, then **Manager1** is sent **compute-conflict-resolution**which results in selecting[14] p1's instantiation leaving p2, p3 and p4 instantiations in the conflict-set. As a result, p1 is fired by executing its **actions** (attribute of p1) as follows:

1. Executing the first action: (make c2 ^c21 3 ^c22 5)

   This is initiated by obtaining a new timestamp (**retrieve-wme-time**), call it T_2, and then inserting this WM element into **WM-vector** (**insert-wme**) at index T_2. The c2 WM-Distributor sends **match-insert** message to p1-c2-2 (**broadcast-WM-change**) which results in successful constant tests computations and hence the balanced trees of p1-c2-2 are updated. Later, match-knowledge of p1 and p2 are updated (**update-match-knowledge**). Then, each of these productions executes **test-joinmixed**which shows that each of them is eligible to compute join and since p1-c2-1 is a negative joinable CE-Object, then a **check-removal-of-production** message is sent to **Manager1** to remove p1 and p2 instantiations whose variable bindings are consistent with the join-list (argument) of the newly matched WM-tuple of timestamp T_2.

[14] p1 is selected here because it has the highest rating (i.e. 6) over p2, p3 and p4 who all have the same timestamp (i.e. T_1) as p1.

Examining the conflict-set shows that there does not exist any p1 instantiations but there is an instantiation of p2 which has the list of variable bindings ((x .5)). Computations of consistent variable bindings of this list and the join-list of T_2 (((x . (equal 5)))) are successful and hence this instantiation of p2 is removed from the conflict-set.

2. Executing the second action: (make c3 ^c31 50 ^c32 100)

   This is initiated by obtaining a new timestamp (retrieve-wme-time), call it T_3, and then inserting this WM element into WM-vector (insert-wme) at index T_3. The c3 WM-Distributor sends match-insert message to p1-c3-3 (broadcast-WM-change) which results in successful constant tests computations and hence the balanced tree of p1-c3-3 is updated. Later, match-knowledge of p1-c3-3 productions p1 and p3 are updated (update-match-knowledge). Then, p1 and p3 send Manager1 remove-from-cs-production message to remove their respective instantiations from the conflict-set, if any.

   Examining the conflict-set reveals that there are no p1 instantiations but there exists an instantiation of p3 which is then removed in consequence.

3. Executing the third action: (remove 1)

   This entails removing the WM element of timestamp T_1, the WM element matched by p1-c1-1. In consequence, the following steps are executed:

   (a) Send broadcast-wm-change message to the WM-Distributor, c1, which in turn sends remove messages to all its CE-Objects. Since p1-c1-1 is the only CE-Object in the CE-Objects-list attribute of c1, then a remove message is sent to p1-c1-1.

   (b) When p1-c1-1 processes remove, then its balanced trees are updated with the removal of the WM-tuple of timestamp T_1. In addition, productions p1 and p2 are asked to update their match-knowledge (update-match-knowledge) of p1-c1-1. Processing is then terminated in that method because this is a case of a removal of a WM element matching a positive CE-Object and

control is returned to the `broadcast-wm-change` method.

(c) At this stage, `c1` sends `remove-from-cs-timestamp` to `Manager1` to remove production instantiations from the conflict-set who have T_1 timestamp. In this case, there is only one production instantiation that has this timestamp, namely `p4` instantiation which is removed from the conflict-set in consequence.

Since no more actions are to be executed while firing `p1` and no production instantiations exist in the conflict-set of `Manager1`, then execution is terminated.

3.2.4 Interface and visibility of objects

Following Booch's methodology [1] we next need to: (i) establish the visibility of each object to others and (ii) establish the interface of each object to the others. The purpose of these steps is to examine dependencies among objects or classes of objects in an object-oriented system. To understand this dependency better, a graphical representation is borrowed from Booch's methodology but with a small enhancement which clarifies one aspect of object interaction. An ellipse represents a class of objects, a solid arrow indicates a message requesting the execution of a suffered method and a dotted arrow indicates a message requesting the execution of a required method. Moreover, a label on an arrow represents a method that is either required of an object or suffered by it. Figure 3-7 shows interaction between classes of objects in the model of object-oriented execution of OPS5 programs.

3.2.5 Implementation

With the four stages of the methodology complete it is possible to consider the implementation. We have constructed a representation of the classes of objects and their interface using ΤΕΛΟΣ, which is the EuLisp Object System [34]. The implementation is sufficiently complete to allow the running of a number of production system applications, and to experiment with some of the parallel processing facilities of Eulisp as will be discussed in chapter 5.

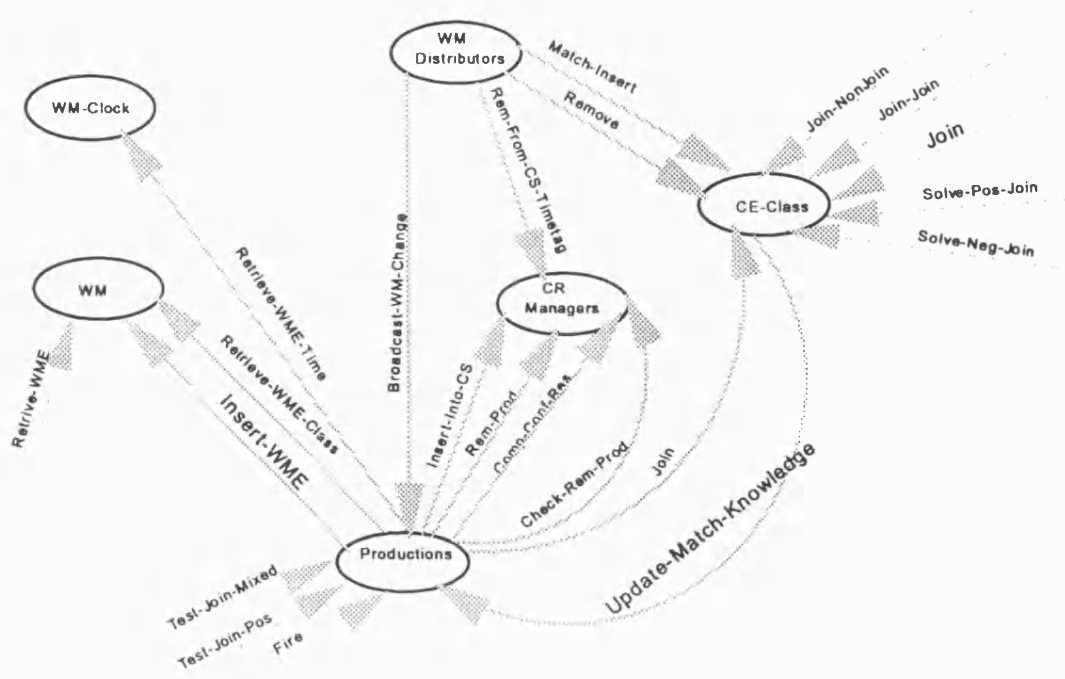

Figure 3-7: Interaction between classes of objects in a transformed Object-Oriented OPS5 program

3.3 Discussion

This approach can be viewed as another example of attempting to improve the performance of Rete-based OPS5 interpreters on uniprocessor platforms. This approach is similar to TREAT [31] in some respects but in some significant respects it differs. On the one hand, it is like TREAT with respect to the following: (i) it processes a removal of a WM element that matches a positive condition element by examining instantiations of the conflict-set that have the timestamp of the WM element which is to be removed, (ii) it does not store intermediate results of join tests between condition elements, which is done using beta memories in the Rete algorithm.

It differs from TREAT in the following ways:

1. The use of object-oriented technology to synthesize an object-oriented inference engine to execute OPS5 production system programs.

2. The distinction between joinable and nonjoinable condition elements resulted in important implications on the execution of OPS5 production systems as will be

discussed in the succeeding steps.

3. It avoids the unnecessary computation of join between condition elements of the same production if a WM element is inserted that matches a negative condition element. The way it is handled here is dependent on whether the CE-Object being matched is joinable or nonjoinable by sending **remove-from-cs-production** or **check-removal-of-production** messages, respectively, to the **CR-Manager**.

4. OOPS5 uses a dynamic ordering strategy at run-time. In that strategy, positive joinable CE-Objects are joined first then negative joinable, if any, and then positive nonjoinable ones, if any. The reason behind computing join with joinable CE-Objects first instead of nonjoinable ones is that if the latter[15] are used, then this may lead to loss of join computations if some or all of joinable CE-Objects fail consistent variable binding computations in succeeding joins. Hence, joining joinable CE-Objects first avoids unnecessary join computations. In addition, joining negative joinable CE-Objects after positive joinable ones is constrained by the semantics of negative joinable CE-Objects which entail that all join variables be resolved before considering a later join.

5. It has the concept of unique[16] CE-Objects. In consequence, we have constructed one implementation with unique CE-objects and one with non-unique CE-objects. The performance of these two schemes is compared in section 4.2.4. However, it is worth mentioning that this concept of uniqueness is at a higher level compared to the sharing of tests in Rete algorithm in the sense that this implementation classifies a condition element as an entity and then looks at its tests, type and joinability.

[15] Recall that nonjoinable CE-Objects compute dummy join.

[16] Recall that a CE-Object is said to be unique if there does not exist any other CE-Object that has the same CE-tests-list, same type (*i.e.* positive or negative) and same joinability

3.4 Conclusion

We have reported our experience in applying Booch's methodology for object-oriented design to the reconstruction of a program based on an analysis of the behaviour required to execute production rule systems. The methodology was easy to use in practice and matches with the natural way of looking at problems from an object-oriented perspective.

Booch does not address explicitly questions of concurrency, although aspects of concurrency are present in the dynamic behaviour of objects. Nevertheless, the methodology assisted in the design and development of a *concurrent* object-oriented platform for executing OPS5 production systems—but only after the introduction of timestamps into the model to control the order of operations.

Object-oriented technology has made it easier to develop a better solution to executing OPS5 rule sets. OOPS5 is better because

- It can add rules and match them with current configuration of WM elements at run-time, which previously required a total recompilation of the network in Rete. This will encourage the construction of reflective systems.

- It allows more efficient handling of adding a WM element that matches a negative CE-object. TREAT changes the condition element into a positive one, recomputing the join and then examining the conflict set for production instantiations that have the same timestamp as the WM element being added. Rete requires the recomputation of consistent variable bindings of the WM element, which requires examination of the relevant parts of the network, but does not examine the conflict set directly.

- It supports the writing of right hand side actions, other than WM or I/O actions, as methods related to productions instead of functions as in OPS5. Hence, OOPS5 is more comprehensive with respect to regarding a production describing a chunk of knowledge.

Chapter 4

Performance of OOPS5

The object-oriented transformation and execution of OPS5 production system programs presented in the previous chapter has been implemented using EuLisp, which is an object-oriented language [34]. To test and evaluate this implementation, four well-known OPS5 production system programs have been transformed and executed successfully using this new approach.

This chapter describes the characteristics of these programs when turned into an object-oriented form and then executed in that environment. In the first part of this chapter, the static measurements which are obtained from the object-oriented transformation phase are presented in detail. The second part of this chapter describes the execution behaviour of these systems in an object-oriented environment. Finally, the last part presents a summary of both static and dynamic measurements.

4.1 Static Measurements

Static measurements are concerned with static information which is gathered during the transformation of OPS5 production system programs into object-oriented ones. However, some of this information is extracted from OPS5 PS programs directly and the rest comes from the object-oriented approach described in the previous chapter.

First, the four OPS5 production system programs that have been used in evaluating this new approach and gathering measurements are briefly described below in ascending

order of the number of objects that are the product of the transformation phase:

1. **Monkey and Bananas (MAB):** MAB is a small and well-known OPS5 production system program. The task of MAB is that when given a description of objects and their locations in a room, it produces a sequence of instructions to the monkey in order for it to grab the bananas. MAB consists of 53 objects when transformed using unique CE-Objects strategy described previously and 73 objects otherwise.

2. **WALTZ:** The task of WALTZ is that when given a two dimensional picture of blocks of World image, it produces a three dimensional labelling information based on Waltz's original method of constraint propagation [45]. WALTZ is comprised of 99 objects when transformed using unique CE-Objects strategy and 175 objects otherwise.

3. **MAPPER:** The task of MAPPER is that when given a starting and ending locations in the city of New York in the USA, along with some constraints such as means of travel (e.g. subway) number of blocks to walk, weather and time, MAPPER produces a suggested optimal way of travelling. MAPPER takes 418 objects when transformed using unique CE-Objects strategy and 570 objects otherwise.

4. **RUBIK:** The task of RUBIK is that when given a certain configuration of the RUBIK cube, it produces a series of instructions which when followed, will restore the RUBIK cube to the right state where each face of the cube has only one colour. RUBIK takes 430 objects when transformed using unique CE-Objects and 707 objects otherwise.

The following sections present the various static characteristics of these four OPS5 production system programs when transformed into object-oriented ones.

4.1.1 Distribution of Objects

When an OPS5 production system program is transformed into an object-oriented one using the approach detailed in the previous chapter, the total number of objects that

Table 4.1: Distribution of Objects

Object Type	MAB	WALTZ	MAPPER	RUBIK
Productions	19	34	117	70
Unique CE-Objects	27	57	240	376
Non-Unique CE-Objects	47	131	392	647
% of Unique/Non-Unique CE-Object	57.45%	43.51%	61.22%	58.11%
WM-Distributors	4	5	58	11
CR-Manager	1	1	1	1
WM	1	1	1	1
WM-Clock	1	1	1	1
Total with Unique CE-Objects	53	99	418	460

is to be obtained is the sum of the number of production objects, CE-Objects, WM-Distributors, Conflict-Resolution Manager, WM and WM-Clock. Table 4.1 presents a breakdown of the objects for the four OPS5 production system programs described above.

One very important observation of this distribution is that the ratio of the number of unique[1] CE-Objects compared to the number of non-unique ones is considerably smaller for this sample of four programs. The impact of utilising the unique CE-Objects strategy should have a significant effect on the performance for the following reasons:

- Saving of redundant computations of constant tests for redundant CE-Objects.
- Saving of redundant updates on WM-AVL-trees for the redundant CE-Objects.
- Saving of redundant updates on AVL trees for each join variable in the `CE-AVL-trees` attribute of the redundant CE-Objects.

These observations reinforces the insights which led to the Rete algorithm, in particular the sharing of constant tests and memory nodes. However, OOPS5 possesses a conservative approach with respect to sharing which is attributable to the information hiding characteristic of object-oriented systems and hence this allows more room for concurrency.

[1] Recall that a CE-Object is said to be unique if if there does not exist any other CE-Object that has the same tests in `CE-tests-list` attribute, same type and same joinability.

Table 4.2: Distribution of Production Objects with respect to Type and Joinability

Type	MAB	WALTZ	MAPPER	RUBIK
Positive	100%	76%	63%	57%
Mixed	0	24%	37%	43%
Joinable	15	24	49	65
Nonjoinable	4	10	68	5

4.1.2 Distribution of Production Objects with respect to Type

Table 4.2 shows the breakdown of the percentages of productions with respect to type whether being positive or mixed and joinability as being joinable or nonjoinable. The figures in this table show that the number of positive productions is higher than the mixed ones and is sometimes the only one. Hence, this strengthens the decision behind the type of classification which led to assign the two methods `test-join-positive` `test-join-mixed` for positive and mixed production objects, respectively. As a result, this leads us to investigate deeper into the controllers of these classifications and more specifically the CE-Objects distribution with respect to productions as either being positive, negative, joinable, nonjoinable, positive joinable, positive nonjoinable, negative joinable and negative nonjoinable. The first part of table 4.3 shows the average number of CE-Objects of these categories per production object, whereas the second part shows the percentages of theses categories of CE-Objects over the total CE-Objects for the test suite.

From this table, the following remarks were deduced :

- MAB, WALTZ and MAPPER tend to have fewer of these classifications of CE-Objects compared to RUBIK. This shows that RUBIK has more complex left hand sides than other systems and hence it may be concluded that production system programs display variation in complexity of the left hand sides in general.

- Most CE-Objects of these systems are positive and hence this leads us to conjecture that most CE-Objects in production systems are positive.

- Most CE-Objects are joinable except in the case of MAPPER which has almost a balance between joinable and nonjoinable ones. Joinability of CE-Objects is an

Table 4.3: Distribution of CE-Objects with respect to Type

Type	MAB	WALTZ	MAPPER	RUBIK
Positive	2.47	3.62	2.92	8.29
Negative	0	0.24	0.43	0.96
Joinable	2.05	2.68	1.53	8.66
Non-Joinable	0.42	1.18	1.74	0.56
Positive Joinable	2.05	2.53	1.27	7.8
Positive Nonjoinable	0.42	1.09	1.65	0.47
Negative Joinable	0	0.15	0.26	0.86
Negative NonJoinable	0	0.09	0.09	0.09
Total CE-Objects	2.47	3.85	3.35	9.24
% positive/total	100%	94.02%	87.16%	89.71%
% negative/total	0%	5.98%	12.84%	10.29%
% joinable/total	83%	69.61%	45.67%	93.72%
% nonjoinable/total	17%	30.39%	54.33%	6.28%
% positive joinable/total	83%	65.71%	37.91%	84.42%
% positive nonjoinable/total	17%	28.31%	49.25%	5.08%
% negative joinable/total	0%	3.90%	7.76%	9.31%
% negative nonjoinable/total	0%	2.34%	2.69%	0.97%

important issue in that if a CE-Object is joinable then this may imply that it is less selective (i.e. may have more working memory elements to match it than nonjoinable ones) and hence this has two effects. First, in productions that have high number of joinable CE-Objects, cost of updating their AVL trees will be higher compared to nonjoinable ones. Second, cost of joining these joinable CE-Objects will also be considerable especially if there there large number of working memory elements matching them.

- The number of positive joinable and negative joinable CE-Objects is higher than positive nonjoinable and negative nonjoinable ones, respectively. This strengthens the policy adopted by OOPS5 with respect to the order in which CE-Objects of a production are joined as positive joinable ones are joined to negative joinable ones followed by positive nonjoinable ones. As a result, this reduces the effect of the cross-product[2] effect since the number of WM-tuples used in the join may be

[2] A term used by Gupta [15] to refer to the case where a single token in the left or right memory of

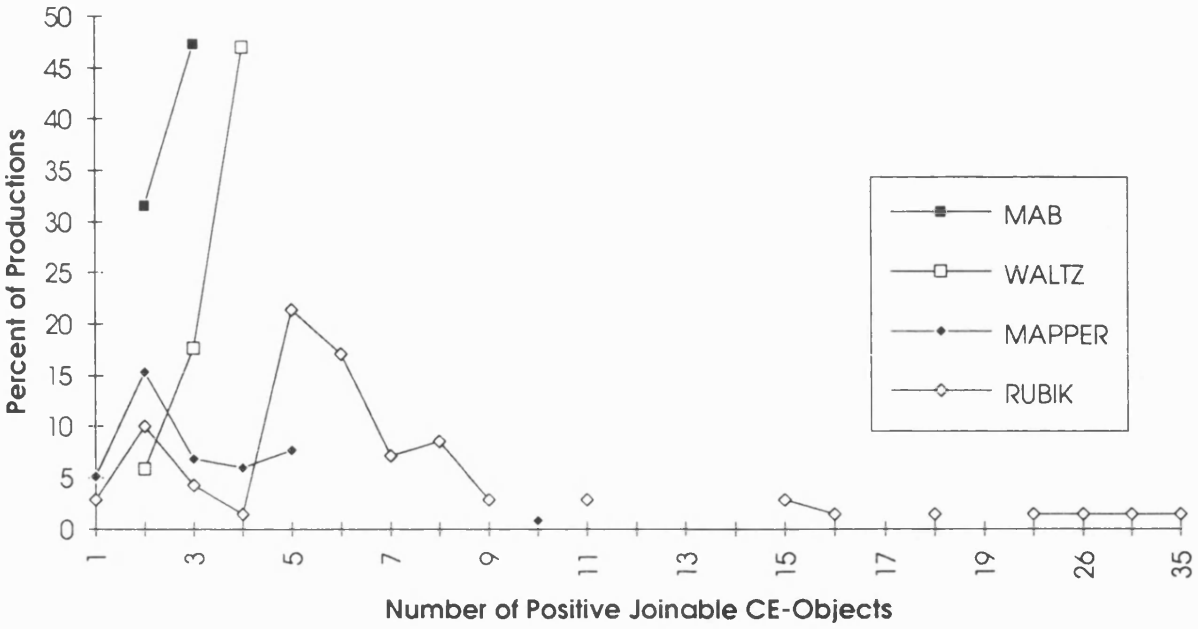

Figure 4-1: Distribution of Positive Joinable CE-Objects over Productions

narrowed down as the join process progresses from one CE-Object to the next.

Although the average value of CE-Objects of each classification per production has proved to be a good indicator for comparing the complexity of the left hand sides of the four production systems in general, it does not seem to be a good indicator of this complexity on the level of the individual production system program. Moreover, this finding was not addressed by Gupta who studied only the breakdown of positive and negative condition elements over production but did not study their joinability (i.e. we believe that the study of the joinability of condition elements is novel). To support this argument, Figures 4-1, 4-2, 4-3 and 4-4 give a closer look at the complexity of the left hand sides of productions in these systems. These Figures show the percentages of positive joinable, positive nonjoinable, negative joinable and negative nonjoinable CE-Objects per production. One valuable observation with respect to RUBIK is that some small percentage of productions have considerably high number of positive joinable (e.g.

---

a two-input node in the Rete network finds many tokens in the right or left memory, respectively. In OOPS5, this refers to a WM-tuple matching a CE-Object that joins with many WM-tuples that match the next CE-Object in the join sequence.

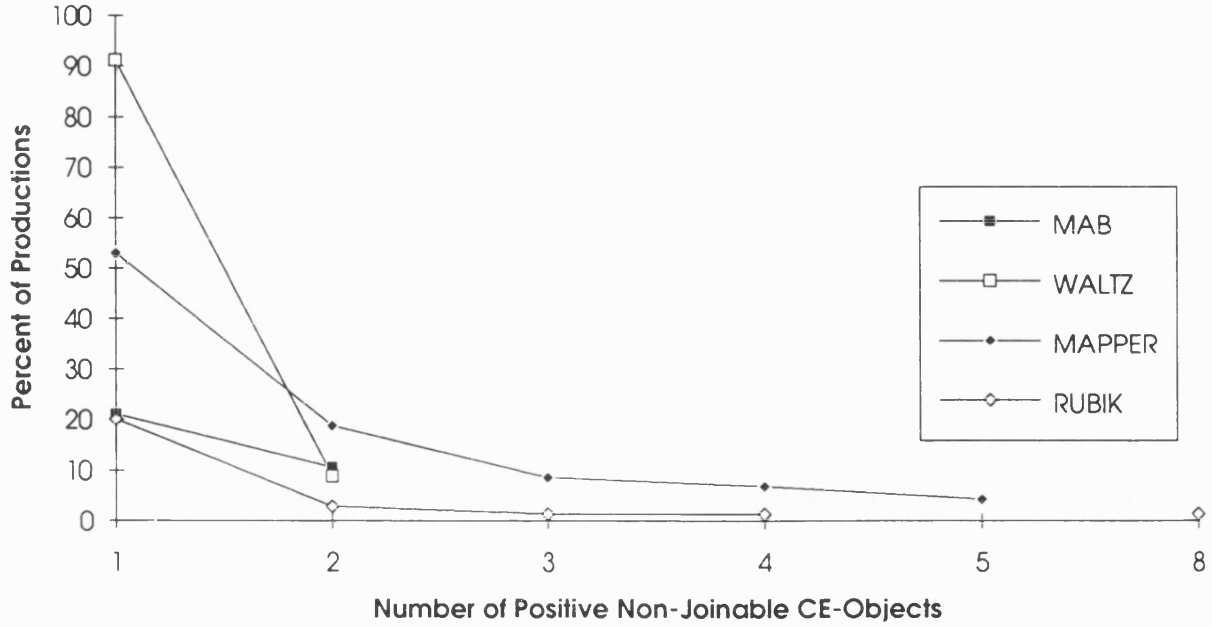

Figure 4-2: Distribution of Positive NonJoinable CE-Objects over Productions

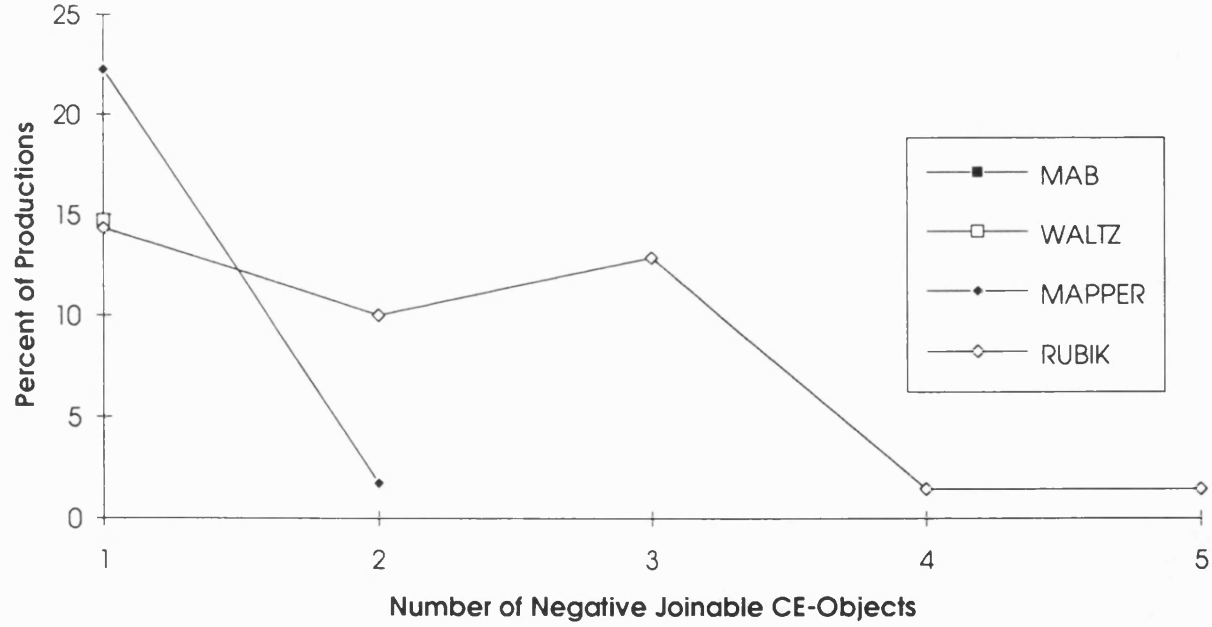

Figure 4-3: Distribution of Negative Joinable CE-Objects over Productions

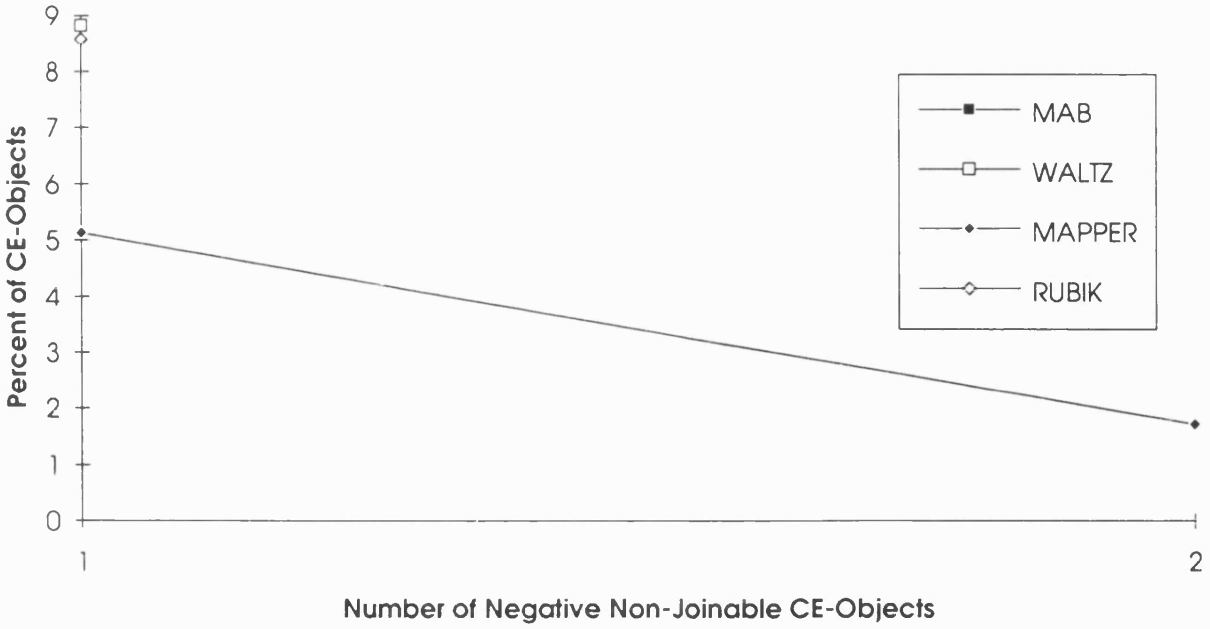

Figure 4-4: Distribution of Negative NonJoinable CE-Objects over Productions

15, 16, 26, 35) and negative joinable CE-Objects. It is these CE-Objects that will have an impact on performance during a production system cycle. Also, it is these types of productions that have higher processing requirements compared to other productions if considered during a cycle.

4.1.3 Distribution of Constant Tests over CE-Objects

Figure 4-5 shows the number of constant tests (i.e. stored in ce-tests-list attribute of a CE-Object) per CE-Object. Although computations of constant tests is cheap compared to the other computations in the execution process, it is worth observing that most CE-Objects tend to have one to two constant tests for the four production system programs except in some small percentages of CE-Objects in RUBIK.

4.1.4 Distribution of AVL Trees over CE-Objects

Figure 4-6 shows the distribution of the number of AVL trees with respect to joinable CE-Objects. These AVL trees[3] are the ones that are used to store bindings of join

[3]These AVL trees are stored in the CE-AVL-trees attribute of a CE-Object.

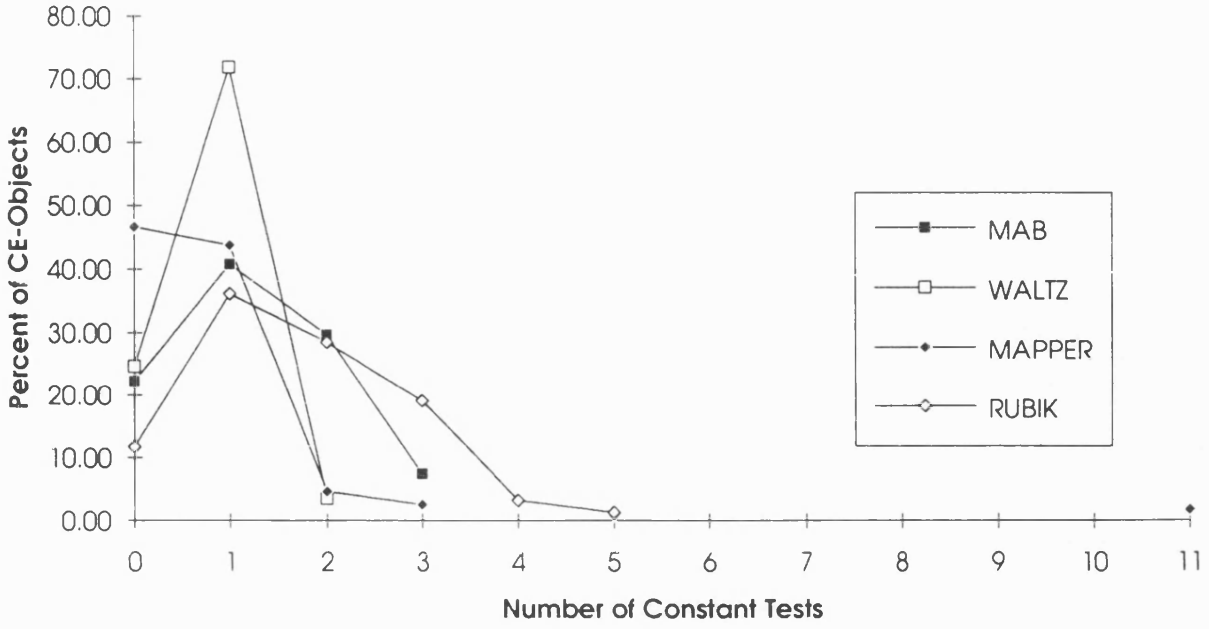

Figure 4-5: Distribution of Constant Tests over CE-Objects

variables as a result of a working memory element has been successfully matched by a joinable CE-Object. The number of such AVL trees reflect the complexity of joinable CE-Objects and the impact on performance for updates of these AVL trees.

The graph in this Figure shows that the majority of CE-Objects have between two to three AVL trees and hence this strengthens the decision of using AVL trees to store bindings of join variables. Moreover, on average there is 0.96, 1.65, 1.43 and 1.82 AVL trees per CE-Object for the four production systems. This may lead us to conjecture that CE-Objects tend to have low number of AVL trees in general.

4.1.5 Distribution of Actions over Productions

For the purpose of efficiency of production systems, the types of actions that have been investigated are the ones that deal with addition, modification and deletion of a working memory element. However, modifying a working memory element is the spawning of a removal of a working memory element and then an insertion of a new one. Hence, studying the distribution of actions reduces to actions that deal with addition and deletion of working memory elements.

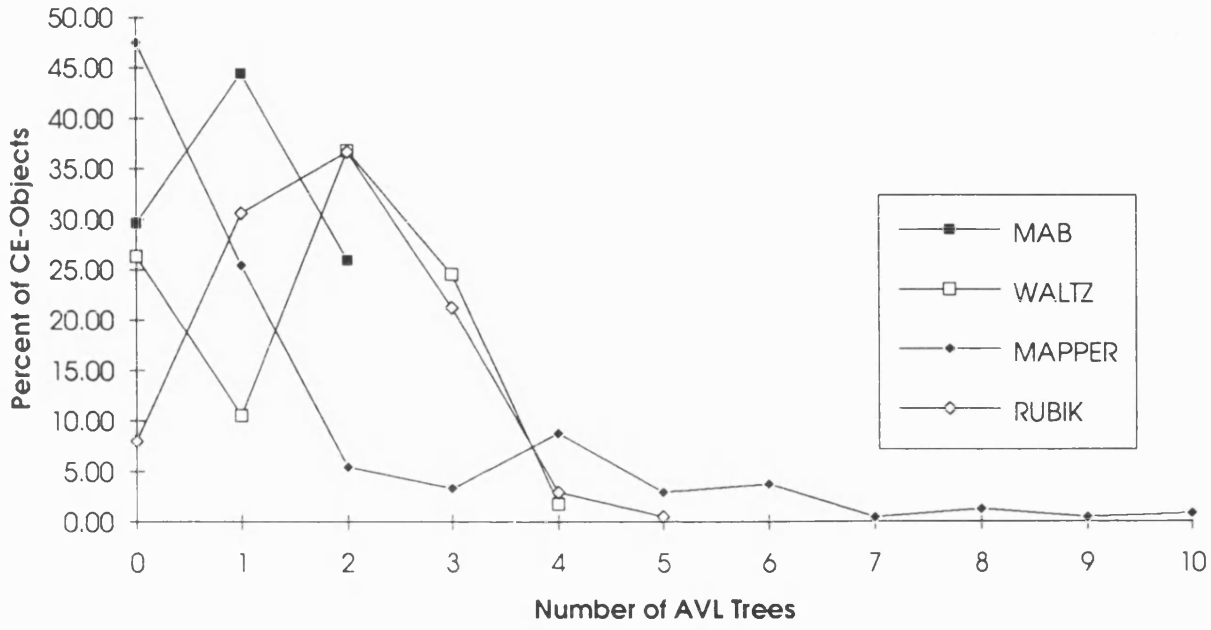

Figure 4-6: Distribution of AVL Trees over CE-Objects

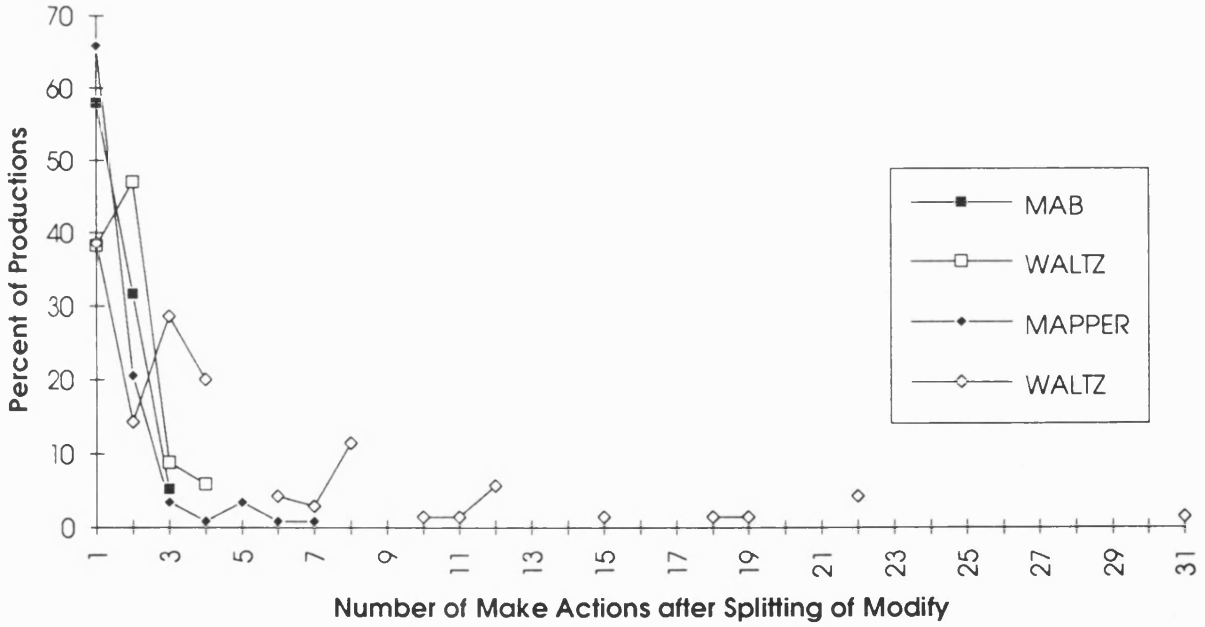

Figure 4-7: Distribution of Make after splitting of Modify

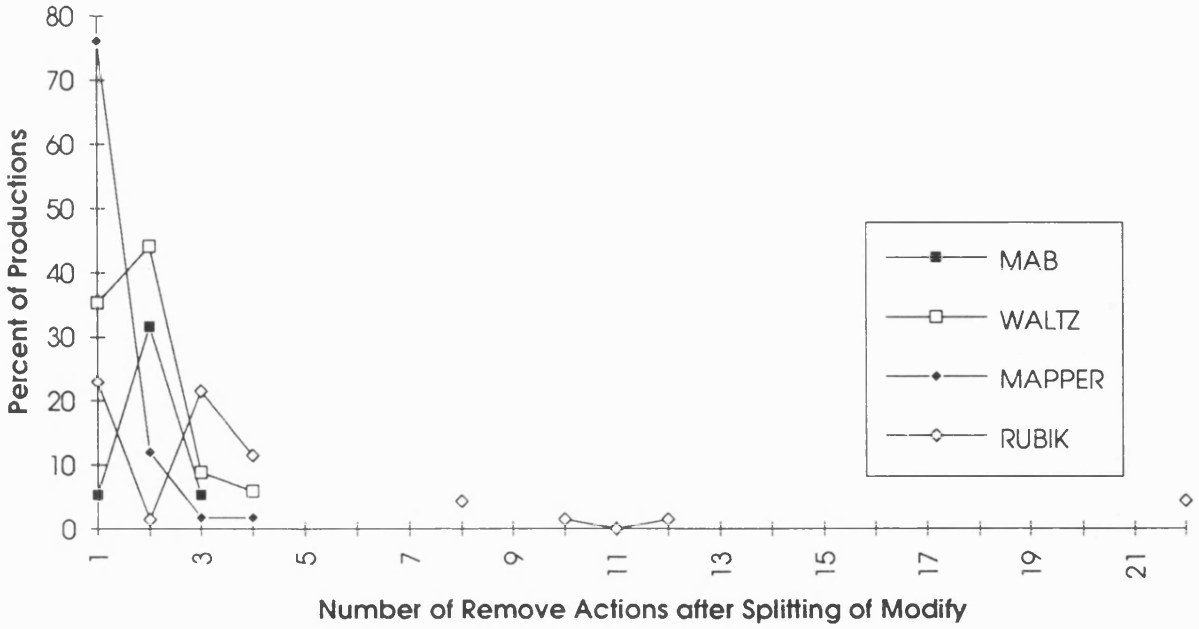

Figure 4-8: Distribution of Remove after splitting of Modify

Figure 4-7 and 4-8 show the distribution of actions that add and delete working memory elements, respectively for the four production systems. On average these production systems have 1.37, 1.82, 1.49 and 11.19 actions that add working memory elements and 0.84, 1.74, 1.12 and 2.96 actions that delete working memory elements per production, respectively. RUBIK has higher number of both types of actions on average compared to the other systems and, more specifically, some of its productions have a very high number of these actions (e.g. 31 addition and 22 deletion actions). Productions that have such high number of actions have the following implications if they get fired at some stage:

1. Addition or deletion of working memory elements that match joinable CE-Objects would imply high number of updates on respective AVL trees of CE-Objects.

2. High number of additions of working memory elements that match negative CE-Objects benefits from the policy adopted in OOPS5 only by removing respective productions instantiations from the conflict-set if any[4].

[4] see `remove-from-cs-production` and `check-removal-of-production` methods discussed in the last

3. Likewise, a high number of deletions of working memory elements that match positive CE-Objects also benefits from the approach taken by OOPS5 which is similar to the TREAT algorithm in removing production instantiations that have the timestamp of the WM just added[5].

In summary, although RUBIK has the highest number of these actions compared to other systems, one may conjecture that production systems tend to have fewer actions that delete working memory elements than those that add them. This strengthens the approach followed in this research with respect to addition of working memory elements that match negative CE-Objects which is handled through direct examination of the conflict-set.

4.2 Dynamic Measurements:

Dynamic measurements refer to measurements that are gathered during the execution of the object-oriented transformed OPS5 production system programs. One of the advantages of this approach is the ease of monitoring the behaviour of such systems.

4.2.1 State Transitions of Production Objects

Studying the behaviour of production objects right from the beginning of the firing of a new production system cycle, a production object encounters some state transitions in that cycle. In this section, first these states are described in a manner to show how a production object transfers from one state to another in order to be inserted into the conflict-set attribute of CR-Manager object and then a detailed analysis is presented on the state transitions of productions objects during the execution of the four production systems:

Static State: Initially and right before the firing of a new cycle, every production object is said to be in static state and hence this set is referred to as the Static Productions (SP) set.

---

chapter for production whose negative nonjoinable and negative joinable CE-Objects get matched by a working memory element, respectively.

[5]see remove-from-cs-timestamp method in the last chapter.

Affect State: The concept of referring to the state of production object to be in the affect state is extracted from the definition of the affect-set[6] [15] with the difference here in that after a static production object changes to affect state, it may continue to the next state or stay in the affect state for the duration of a cycle. This new distinction refers to a state rather than a set. The pattern of change of state from static to affect can be easily monitored in OOPS5 by following the result of sending a `match-insert` message to a `CE-Object` by a corresponding `WM-Distributor` object and as a result it succeeds in computing a match and passes a message to a corresponding production object to update its match-knowledge (i.e calling `update-match-knowledge` method).

Dormant State: The set of static productions that are not in the affect state are referred to as dormant and they remain dormant for the duration of a cycle.

Join State: A production object is said to be in the join state if it transfers from the affect state and is eligible to compute a join. Recall that from previous chapter that a production object is said to be eligible to compute a join if each of its positive CE-Objects has some working memory element matching it and each of its negative nonjoinable CE-Objects does not have any working memory elements matching it. A production object which executes `test-join-positive` or `test-join-mixed` method but passes these join eligibility conditions is said to be in the join state. The set of productions objects in the join state are referred to as the join set. A tighter definition of the join set definition is used by Miranker in his TREAT algorithm in referring to a production to be active when each of its positive condition elements is matched by at least one working memory element. However, the definition of the join set here is more comprehensive in including negative nonjoinable condition elements and is further extended to denote a state in the duration of a production system cycle.

Conflict State: This is the final state and succeeds the join state to this state. This state reports the success of the join between the CE-Objects of a production in the join state which is triggered in OOPS5 as an insertion of an instantiation of this production object into conflict-set when an `insert-into-conflict-set` message is sent

[6] Recall that a production is said to be affected by a change to working memory if at least one of its condition elements is satisfied by this change [15]

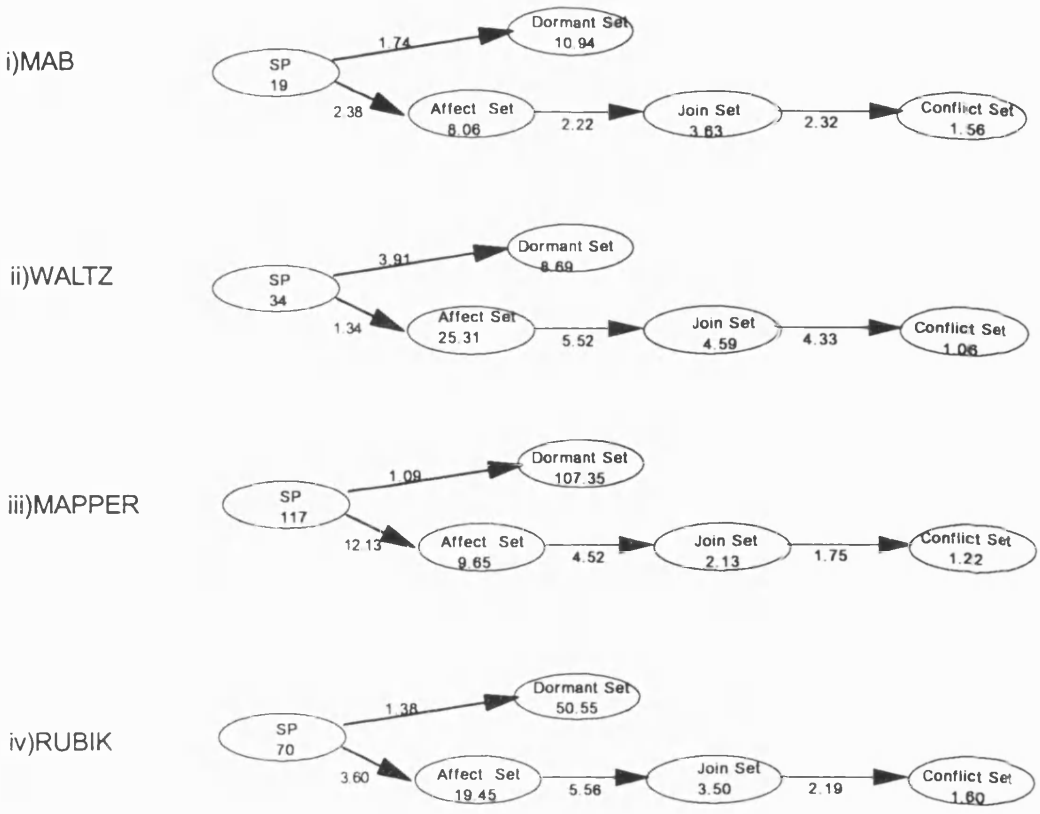

Figure 4-9: State Transition Diagram for MAB, WALTZ, MAPPER and RUBIK

to `CR-Manager` object.

The pattern of change of states is fixed. That is to say that a production changes from static to dormant or from static to affect to join to conflict states. These states were monitored during the execution of the four production systems and two types of presentations for the results are introduced.

First, a state transition diagram showing the average sizes of these sets appears in Figure 4-9, where nodes represent states and arcs between any two states represent a ratio between the average value of productions in predecessor state to average value of productions in the successor state in a cycle.

Second, a plotting of the sizes of the static, dormant, affect, join and conflict sets per cycle is presented in Figures 4-10, 4-11, 4-12 and 4-13 for the four production system programs, respectively.

The following are remarks were deduced from these Figures:

- A conjecture may be made that production system programs that have high num-

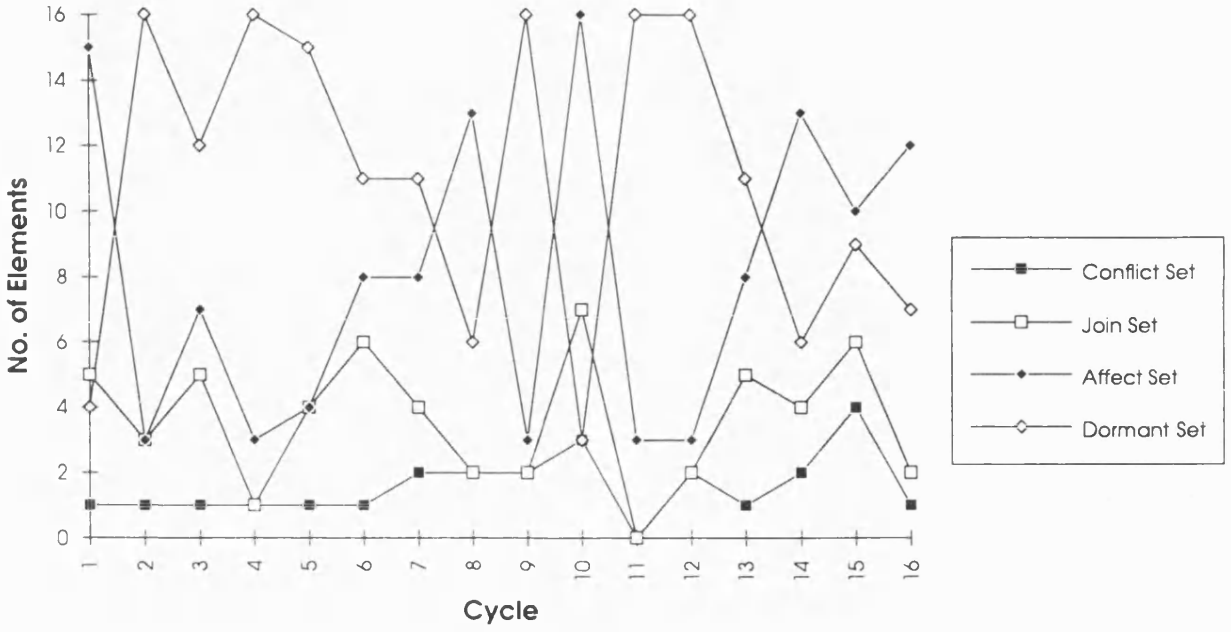

Figure 4-10: State Transitions per Cycle for MAB

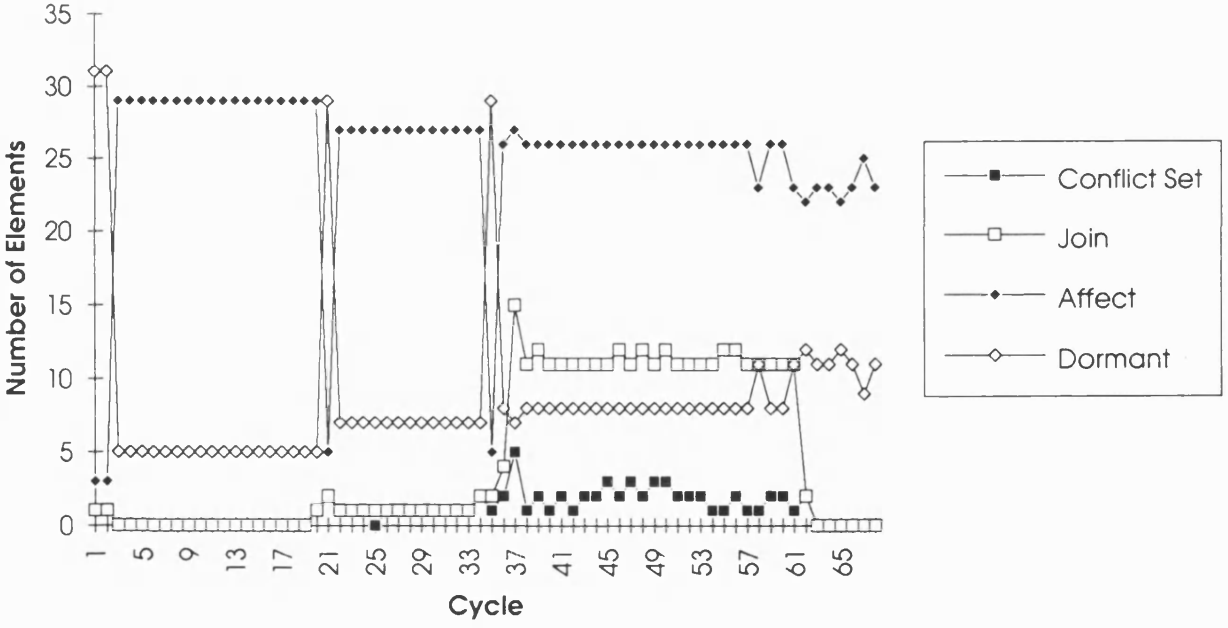

Figure 4-11: State Transitions per Cycle for WALTZ

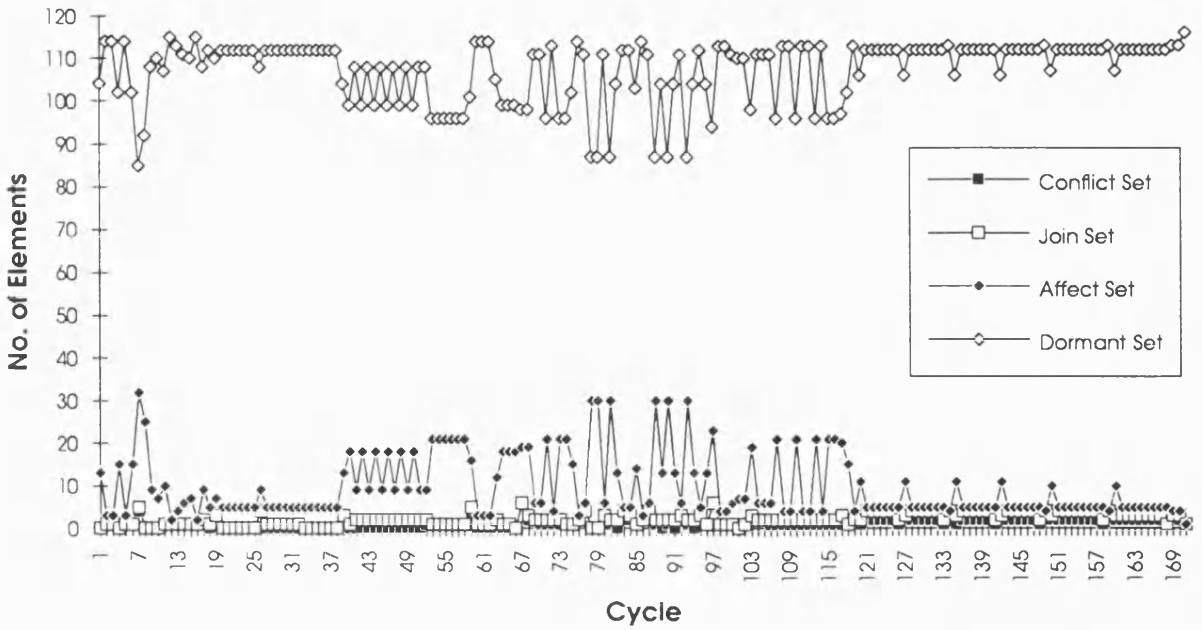

Figure 4-12: State Transitions per Cycle for MAPPER

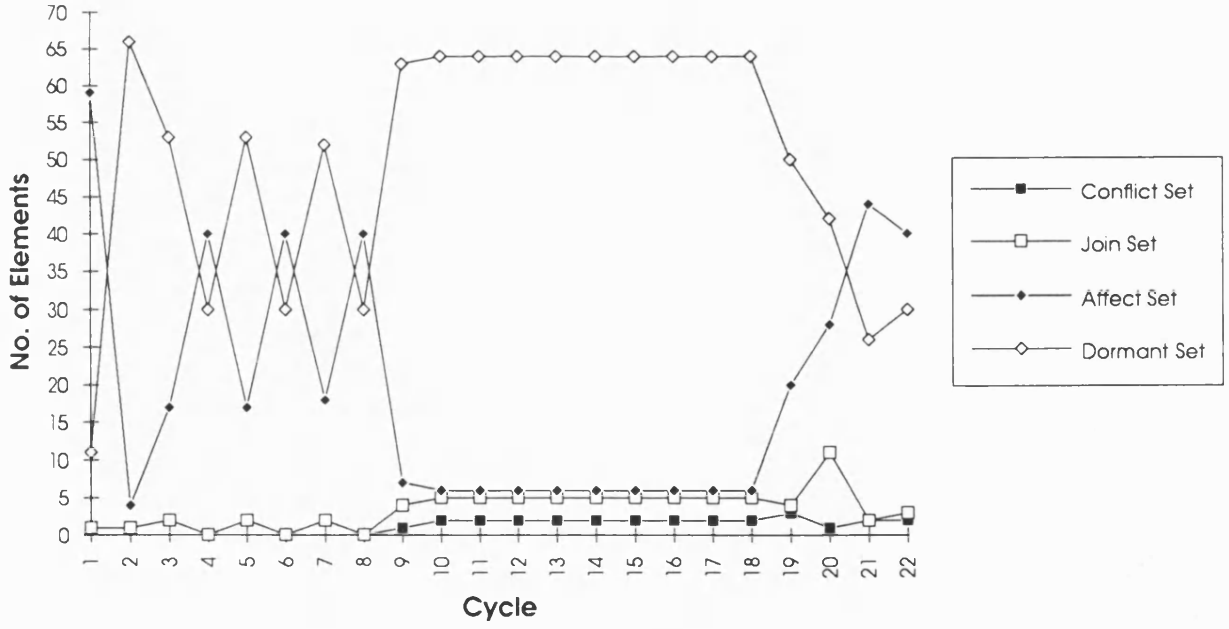

Figure 4-13: State Transitions per Cycle for RUBIK

ber of productions tend to have the majority of its productions in dormant state (e.g. 91.75% and 72.12% for MAPPER and RUBIK, respectively) which implies the opposite of this conjecture for the case of the affect state. Hence, Gupta's observation that the number of affected productions are independent from the total number of productions in a production systems program may have to be amplified to the following statement:

> "The number of affected productions in a production system are independent of the total productions *but systems with many productions may have a smaller affect set than those with a few productions.*"

In fact, this conjecture has been found to be valid for both the production systems being studied here and the six production systems studied by Gupta.

- Considering that the cost of deciding on joinability of a production is not high compared to the cost of joining its CE-Objects and that the size of the join set is small compared to the size of the affect set, this leads us to concentrate on the characteristics of the productions in the join set which utilise most of the cost in a production system cycle as will be discussed in section 4.2.3. Hence, this leads us to conclude that the size of the join set may be a better indicator of production level parallelism than the affect set chosen by Gupta. Consequently, this questions Gupta's reliance on the affect set in choosing the number of processors in the Production System Machine (PSM) he proposed.

- The ratio of join set to conflict set is higher than the ratio of affect set to join set which leads us to say that the join set has closed the gap between the affect set and the conflict set.

- On average, the upper bound on the size of the conflict set for the four production system programs seems to be 2 which implies that searching the conflict set cannot be an expensive operation, especially when considering the heavy dependence of OOPS5 on direct removal of production instantiations from the conflict-set[7].

[7] Recall that this done when a working memory element is removed or added that matches a positive

Table 4.4: Movements into and out of the Conflict-Set

Production System Program	MAB	WALTZ	MAPPER	RUBIK
Insertion	1.56	3.08	2.54	2.64
Positive Removals	0.63	1.8	0.49	0
Negative Join Removals	0	0.28	0.12	0.41
Negative Non-Join Removals	0	0	0.94	0.64
Ratio of Insertions to Removals	2.48	1.48	1.63	2.51

In summary, the state transitions diagram for a production system run is considered to be a valuable performance analysis tool in that it describes briefly the behaviour of production systems at run-time irrespective of the underlying algorithm being employed by the inference engine.

4.2.2 Movements into and out of the Conflict-Set

The conflict-set attribute of the CR-Manager object suffers one insertion and three types of deletions of productions instantiations. The insertion into the conflict-set is done through the `insert-into-conflict-set` method. The three types of deletions[8] are:

Positive Removal: This is the removal of a production instantiation that has got a positive CE-Object matched by working memory element of a certain timestamp.

Negative Join Removal: This is the removal of a production instantiation that has got negative but joinable CE-Object that has already been matched by a newly inserted working memory element.

Negative Non-Join Removal: This is the removal of a production instantiation that has got a negative but non-joinable CE-Object that has already been matched by a newly inserted WM element.

The monitoring of these movements was recorded for the four production system programs on cycle by cycle basis and the results are presented in Figures 4-14, 4-15,

or negative a CE-Object, respectively.

[8]These removals are handled by `remove-from-cs-timestamp`, `check-removal-of-production` and `remove-from-cs-production` methods of the CR-Manager object.

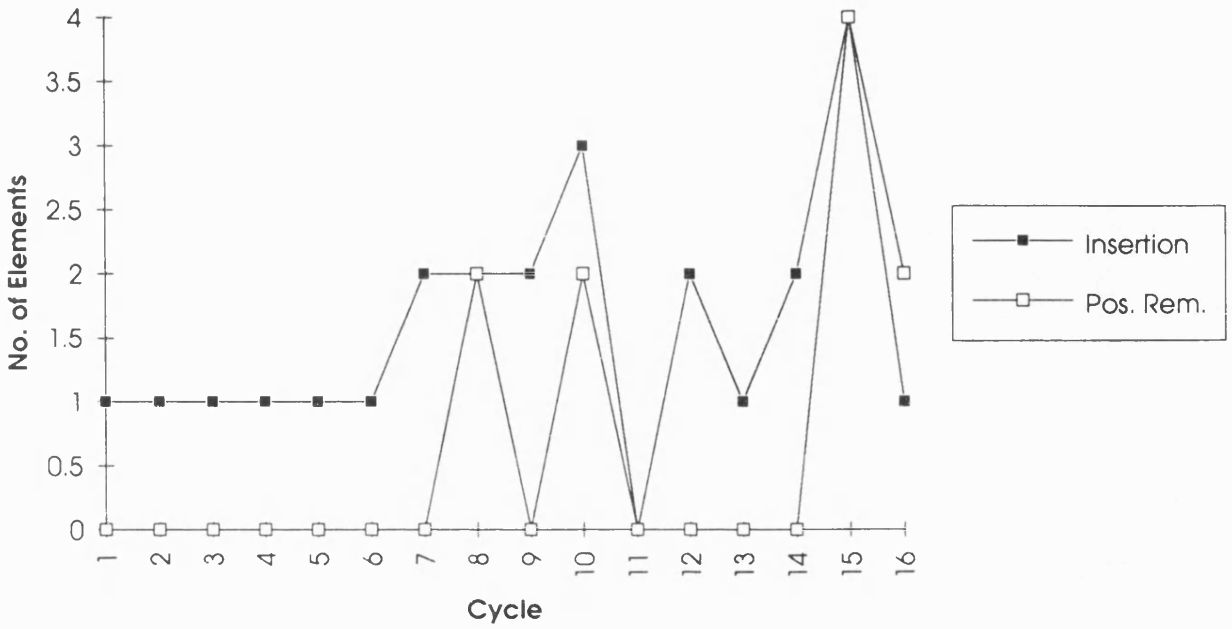

Figure 4-14: Movements into and out of the Conflict-Set in MAB

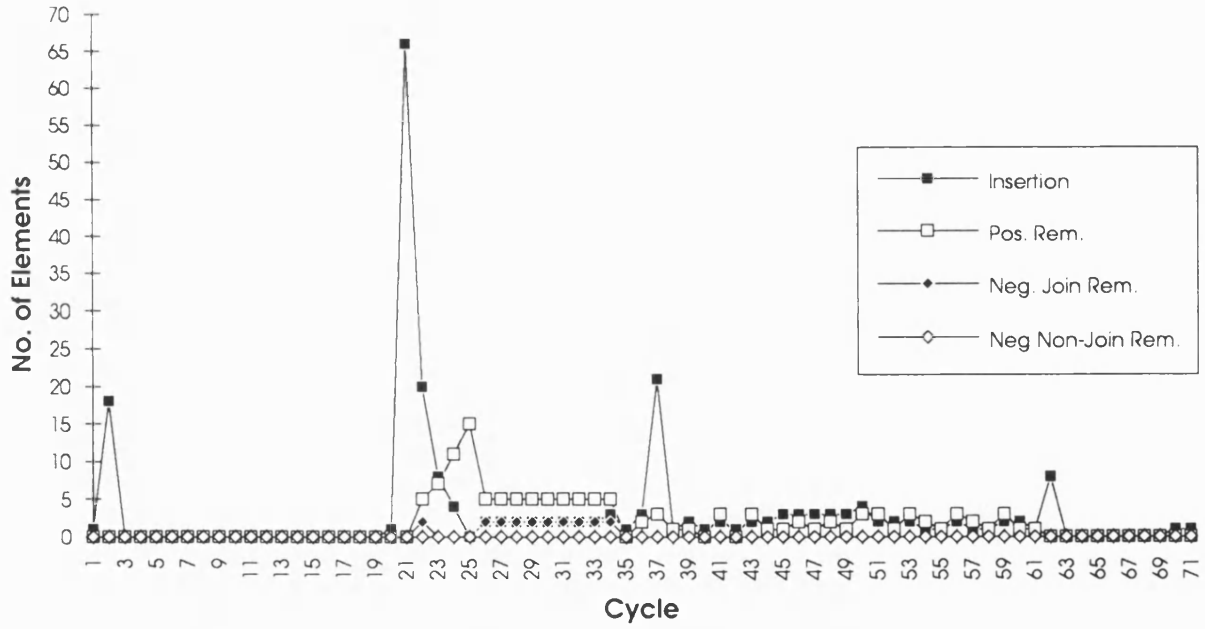

Figure 4-15: Movements into and out of the Conflict-Set in WALTZ

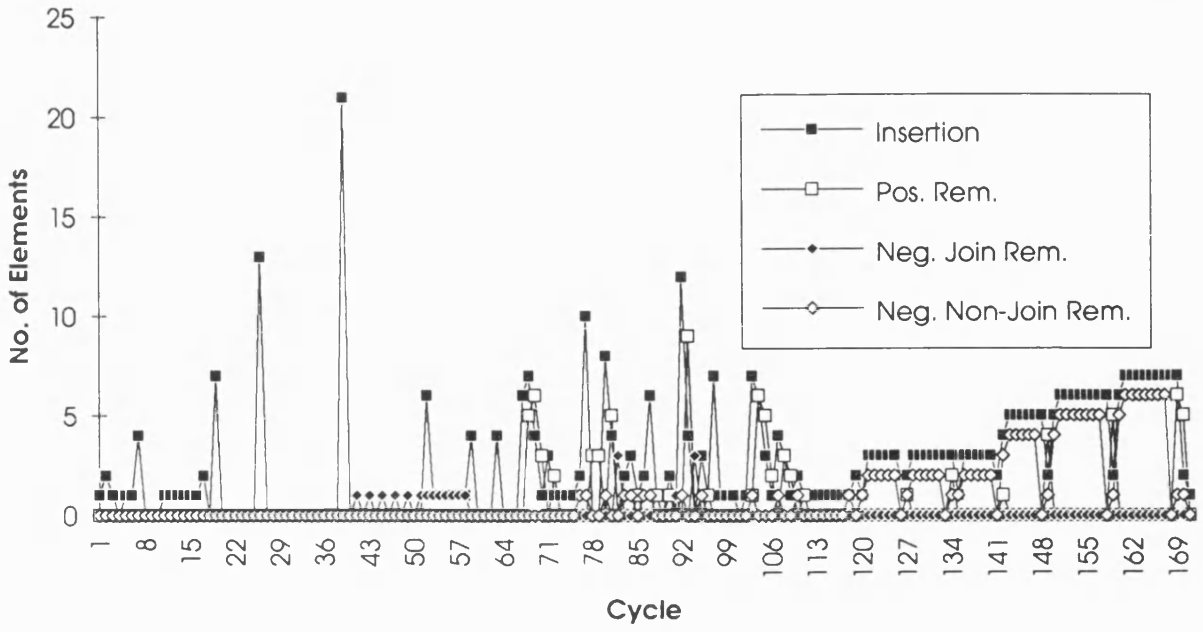

Figure 4-16: Movements into and out of the Conflict-Set in MAPPER

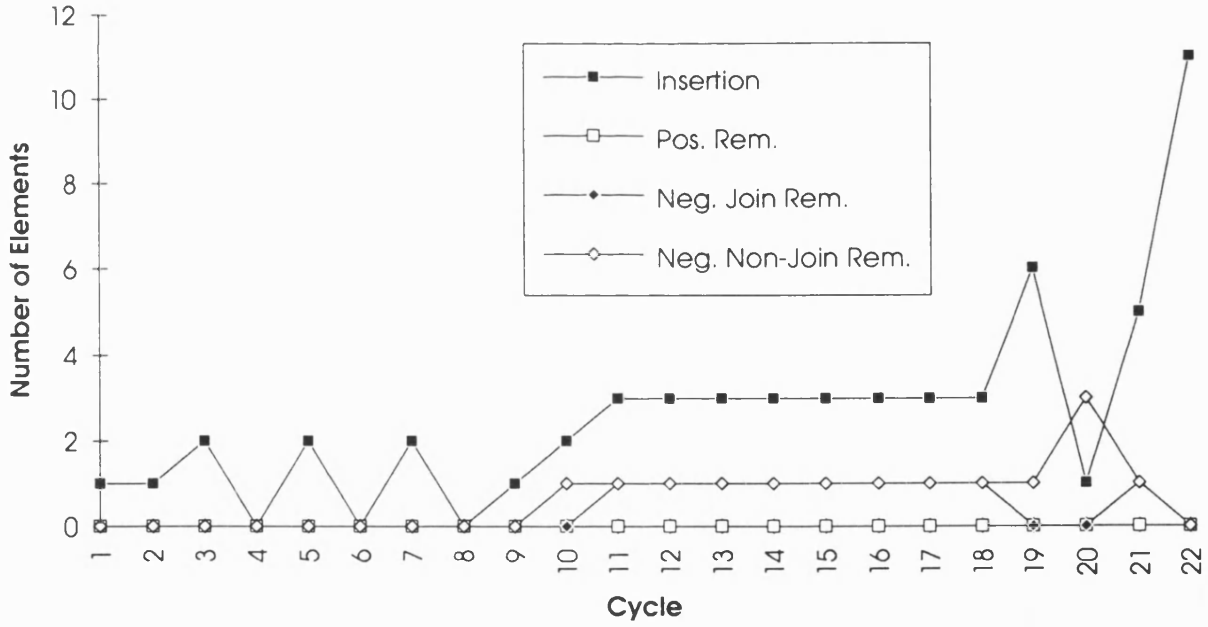

Figure 4-17: Movements into and out of the Conflict-Set in RUBIK

4-16 and 4-17, respectively. Table 4.4 lists the average number of insertions, positive removals, negative join removals and negative non-join removals for these systems which has led to conclude the following:

- The failure of TREAT to handle removal of production instantiations from the conflict-set as a result of the addition of a WM element that matches negative joinable or non-joinable CE-Objects has a high impact on performance if a join computation has to be recomputed as in TREAT and Rete. This impact was found to be a factor of 1.16, 0.33, .20 and 0 over total cost of join for RUBIK, WALTZ, MAPPER and MAB, respectively.

- The average number of negative removals (i.e. join or non-join) is 1.06, 1.05, .028, 0 compared to 0.49, 0, 1.80, 0.63 on average in the case of positive removals for MAPPER, RUBIK, WALTZ and MAB, respectively. This has led us to conclude that production systems tend to have a high potential number of negative removals.

- The average number of negative non-join removals tends to be higher than the negative join ones then provided that the cost associated with the former is less than the latter[9], this supports the approach adopted by OOPS5 with respect to classifying CE-Objects into joinable and nonjoinable.

- The impact on performance had a positive removal been eliminated and replaced by a recomputation of join was found to be a factor of 6.47, 2.64, 0.0074 and 0 over the total cost of join for MAPPER, WALTZ, MAB and RUBIK, respectively. This supports the strategy employed by both OOPS5 and TREAT with respect to positive removals for MAPPER and WALTZ.

- The ratio of insertion into and removal from the conflict-set (see line 5 in table 4.4) has been discovered to be a valuable figure since it gives the programmer an indication of unnecessary production instantiations. If this ratio is high, then

[9]Recall that negative join removal requires checking consistent variable bindings between the join-list of a production instantiation and the join-list of a negative CE-Object being matched by a working memory element (see `check-removal-of-production` method of CR-Manager object in section 3.2.2)

this suggests that the programmer should reduce this ratio by making productions more specific by either adding more condition elements to such productions or putting more restrictions on the tests in a condition element (e.g. using more predicates to narrow selectivity of matching working memory elements). Hence, this ratio is an important finding with respect to improving performance of production systems in general which was not investigated previously.

In summary, for an inference engine to avoid the recomputation of the join between the CE-Objects of a production as a result of: (i) removing a working memory element that matches a positive CE-Object or (ii) adding a working memory element that matches a negative CE-Object, such an inference engine has to consider direct removal of production instantiations from the conflict-set. Hence, OOPS5's novel solution to the problem is a major advance over the approaches in TREAT and Rete.

Moreover, the ratio of insertion to removal from the conflict-set is an important indicator for improving the performance of real-time production system programs irrespective of the underlying algorithm used by the inference engine.

4.2.3 Analysis of the Cost per Cycle

The analysis of the total cost of running a production system using OOPS5 is analysed based on dividing the cost per cycle into four categories as follows :

1. Computations of constant tests was monitored by summing the total time spent in processing the `match-insert` method prior to updating AVL trees of CE-Objects per cycle.

2. Computations related to updating AVL trees of CE-Objects at both times when inserting or deleting WM-tuples into or from AVL trees of a CE-Object. This type of computation was monitored in `match-insert` and `remove` methods.

3. Computations related to updating match-knowledge of productions per cycle was monitored by summing the cost of processing the `update-match-knowledge` per cycle.

4. Computations related to joining CE-Objects of the same production per cycle which were monitored from the time when the `update-match-knowledge` method hands off computation to `test-join-positive` or `test-join-mixed` methods for positive or (negative or mixed) productions, respectively, until the end of current cycle for that production.

To analyse the cost involved in executing the four production system programs based on the above cost categories, it was best found to consider this analysis on the level of the individual production system program. As an aid in this analysis, a graph is used for each of these systems to show the following:

- Percentage of the total cost of a cycle[10] over total execution time.
- Percentage of the cost of constant tests computations over the total cost in a cycle.
- Percentage of the cost of updating AVL trees over total cost in a cycle.
- Percentage of the cost of updating match-knowledge of productions over the total cost in a cycle.
- Percentage of the cost of join over the total cost in a cycle.

Figure 4-18 shows the different percentages of cost for MAB, from which one can deduce two facts. First, for a small production system program like MAB, it is more likely that the computations of constant tests, AVL trees updates and match-knowledge updates tend to be higher than the cost of join which can be attributed to: (1) small number of actions in the right hand side of a production, (2) small number of WM-tuples stored in the AVL trees of CE-Objects and (3) small number of CE-Objects per production. Second, there is not a great magnitude of change of cost from cycle to cycle.

WALTZ and MAPPER share the same characteristic in that few cycles are responsible for most of the total cost which can be revealed from Figures 4-19 and 4-20 for these two systems, respectively. In the case of WALTZ, 24% of the cycles (i.e. cycles 21

[10]Total cost is assumed to be the sum of the cost of constant tests, AVL trees updates, match-knowledge updates and join computations.

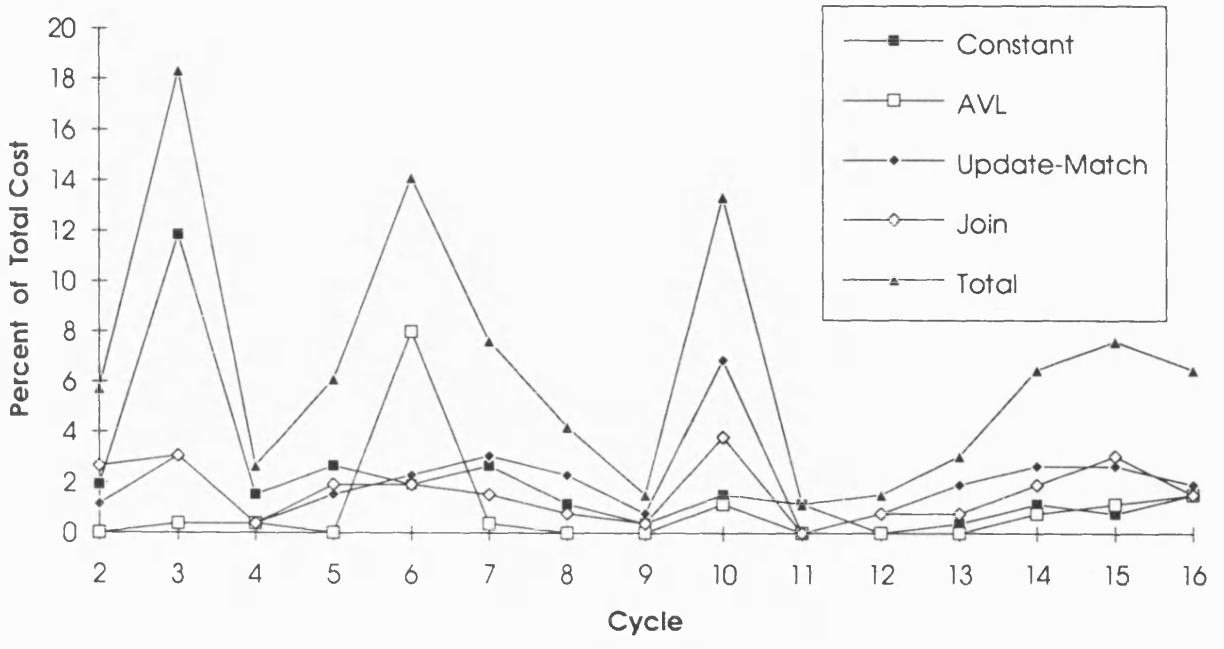

Figure 4-18: Analysis of Cost per Cycle for MAB

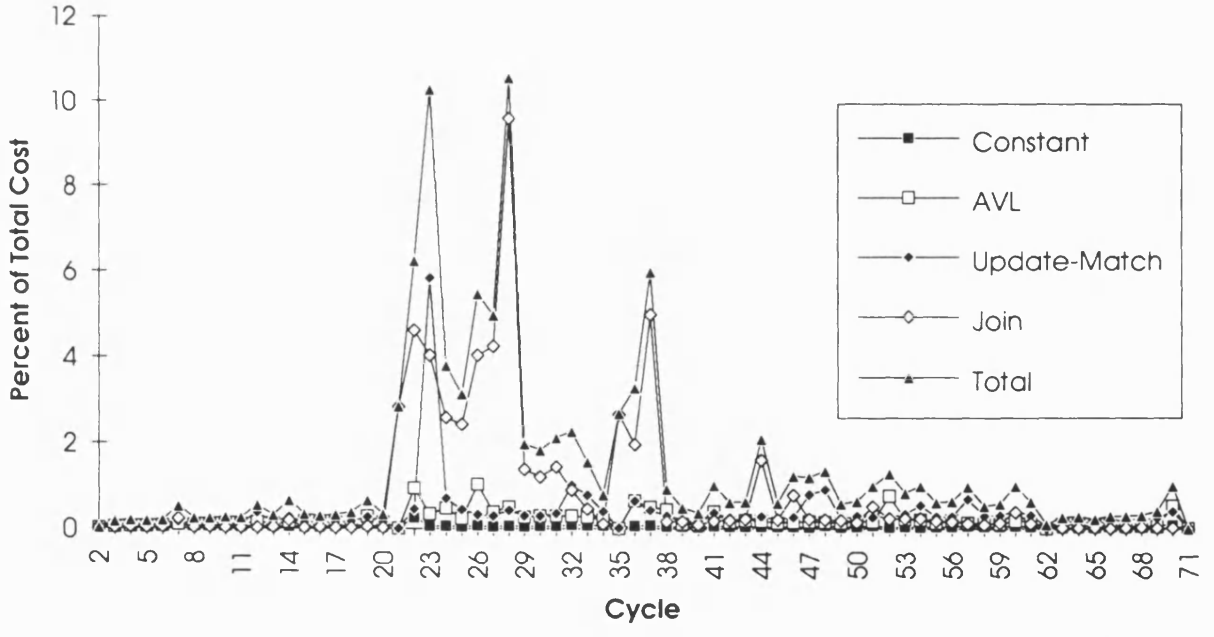

Figure 4-19: Analysis of Cost per Cycle for WALTZ

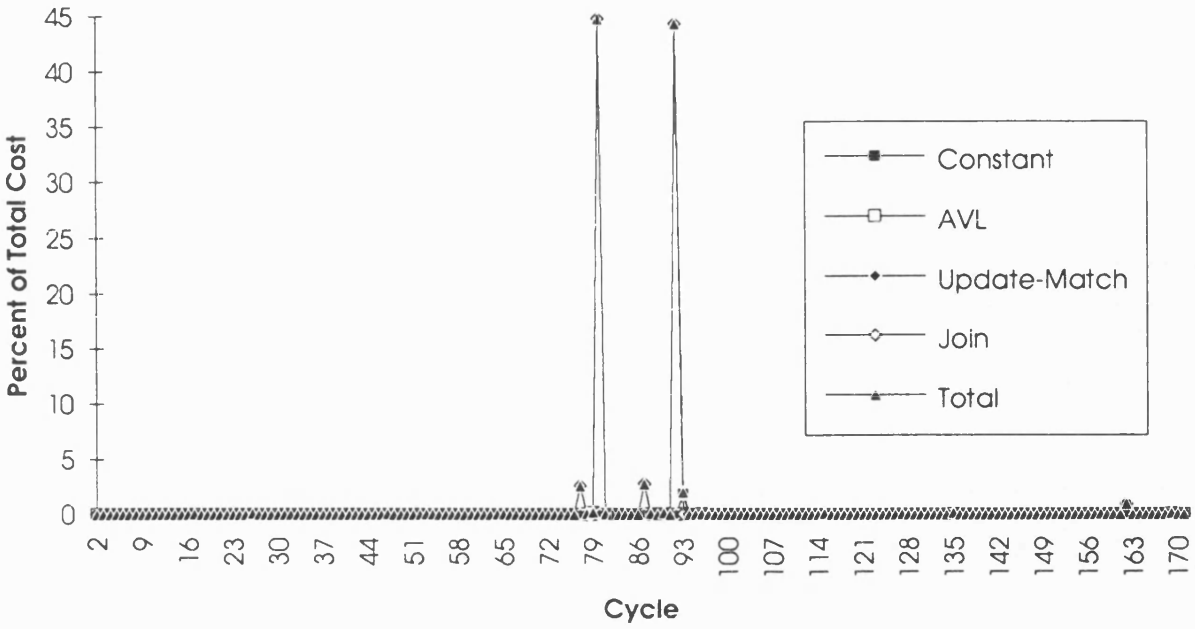

Figure 4-20: Analysis of Cost per Cycle for MAPPER

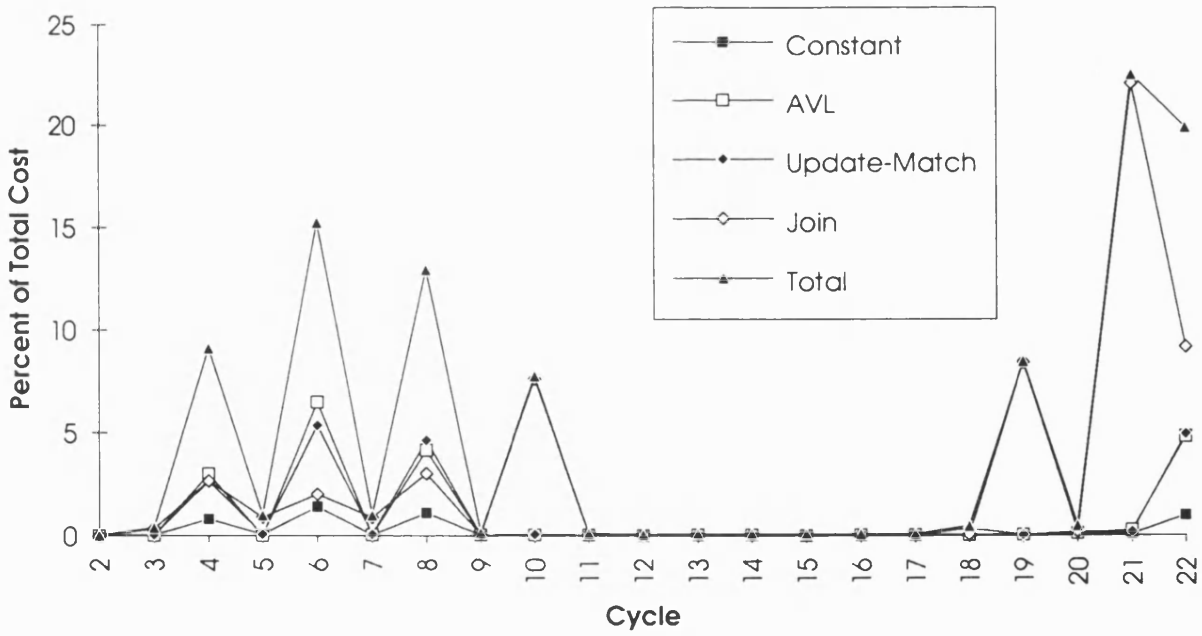

Figure 4-21: Analysis of Cost per Cycle for RUBIK

to 38) is responsible for 70% of the total cost and 3% of the cycles (i.e. cycles 77, 80, 87, 92, 93) is responsible for 96.56% of the total cost in the case of MAPPER. In these few cycles, most of the cost is utilised in computing join. This is due to the fact the number of WM-tuples involved in the join are higher in these cycles compared to the remaining cycles. More specifically, these WM-tuples that matched negative joinable CE-Objects in the case of WALTZ (i.e. 32 WM-tuples on average) whereas in MAPPER they matched positive joinable CE-Objects (e.g. 326 on average in cycle 80). Moreover, the number of these tuples that match negative CE-Objects in the case of WALTZ is much higher than the ones that match positive joinable CE-Objects and vice versa in the case of MAPPER. This leads us to conclude that using parallelism at this level (i.e. join level) may lead to some speed-up by spreading the cost of join over more than one processor and hence this requires formulating a strategy to deal with joining negative and positive CE-Objects as will be discussed in the next chapter.

RUBIK tends to exhibit more complex behaviour than WALTZ and MAPPER as shown in Figure 4-21. Examining this Figure reveals that 30% of the cycles (i.e. cycles 4, 6, 8, 10, 19, 21, 22) are responsible for 95.81% of the total cost. Four of these cycles (i.e. cycles 4,6,8,22) have their cost almost uniformly distributed over AVL trees updates, match-knowledge updates and join computations which when combined together result in 57.17% of the total cost. However, examining the productions involved in these cycles reveals that these productions have a high number of CE-Objects (e.g. 41) which have a high number of AVL trees and large large number of actions in the right hand side. These three factors imply a large number of updates on AVL trees, match-knowledge and high join cost. The other cycles (i.e. 10, 19, 21) which are responsible for 38.64% of the total cost have most of the cost taken in join computations of WM-tuples satisfying positive CE-Objects similar to the case of MAPPER except that the number of WM-tuples is far less in RUBIK (e.g. 10) than in MAPPER.

As a result, to improve on the performance of production systems such RUBIK exploiting action level parallelism combined with join level parallelism may be considered as a remedy. Moreover, this leads us to conclude that OOPS5 and TREAT may not perform better than Rete with respect to production systems that have high number

Table 4.5: Summary of the Total Costs

Production System Program	MAB	WALTZ	MAPPER	RUBIK
Constant Tests	30.53%	3.08%	2.23%	4.84%
AVL trees updates	13.74%	14.31%	0.80%	19%
Match-Knowledge updates	31.30%	25.44%	0.87%	18.65%
Join	24.43%	57.17%	96.1%	57.51%

of condition elements and high number of actions in the left and right hand sides, respectively.

To summarize on the different types of costs involved in executing the four production system programs, table 4.5 presents percentages of these costs over the total cost of execution for theses systems. The first line in the table shows that despite the loss of sharing in OOPS5 compared to high degree of sharing in Rete, the computation of constant tests seems to be bounded by 5% of the total cost as in Rete [11] except in the case of MAB which is a small production system program. The last row shows that the majority of time utilised is involved in join computations except in the case of MAB. The updates on AVL trees and match-knowledge in WALTZ and RUBIK utilised 39.75% and 37.65% of the total cost, respectively compared to 1.67% in MAPPER which reflects the complexity of the left and right hand sides in RUBIK and the responsiveness of CE-Objects in WALTZ to working memory changes.

4.2.4 Results of the Different Implementations of Inference Engines in OPS5

To investigate the possible speed-up that may be obtained using OOPS5 as an inference engine in OPS5, three different implementations of OOPS5 have been used to evaluate the performance of OOPS5 compared to Rete based OPS5. These three implementations are: (1) Relaxing the concept of uniqueness of CE-Objects (2) Using unique CE-Objects and the default ordering of CE-Objects of the same production when carrying out join (3) same as second implementation but ordering of CE-Objects when carrying out join is according to their increasing size of WM-AVL-trees of the

Table 4.6: Execution Results of Different Implementations

Production System Program	MAB	WALTZ	MAPPER	RUBIK
OPS5	1.55	101.62	2041.23	147.35
OOPS5 Non-Unique CE-Objects	2.09	280.80	2236.42	349.48
OOPS5 Unique CE-Objects (not sorted)	1.59	181.50	2171.02	205.25
OOPS5 Unique CE-Objects (sorted)	1.51	165.43	2147.13	172.73
Speed-up using Unique-CE-Objects	1.31	1.55	1.03	1.7
Degree of Uniqueness	1.74	2.30	1.72	1.63
Average Size of Affect Set	8.06	25.31	9.65	19.45

CE-Objects of a production. Table 4.6 gives execution times in seconds for these three implementations (first four lines).

Relaxing the concept of uniqueness of CE-Objects degrades the performance of OOPS5 and is especially observed in production systems that have high degree of uniqueness of CE-Objects[11] and large size of affect set. On the one hand, the higher the degree of uniqueness — which is a static measure — of CE-Objects in a production system, the higher the tendency of that system to save redundant computations of constant tests and updates on AVL trees of CE-Objects had the non-uniqueness approach is taken instead. On the other hand, the size of the affect set — which is a dynamic measure — is supporting evidence for uniqueness. For example, systems like MAB, WALTZ and RUBIK benefited from the uniqueness of CE-Objects approach in obtaining speed-up factors of 1.31, 1.55 and 1.7, respectively, which are higher than the speed-up factor of MAPPER which is attributed to the small size of MAPPER's affect set. Hence, this leads us to conclude that the uniqueness of CE-Objects approach has an impact on improving the performance of OOPS5 and in most cases this impact may be quite significant.

Although we were unable to obtain a TREAT based OPS5 interpreter to run under EuLisp, it is fortunate that Miranker used MAB, WALTZ and MAPPER production system programs for his comparison of TREAT against Rete. Table 4.7 presents the speed-up factors of TREAT[12] over Rete and of OOPS5 over Rete in the cases of both

[11]Obtained by dividing the number of CE-Objects when uniqueness is relaxed by the the number of CE-Objects under the unique strategy.

[12]These factors were calculated by the author of this thesis when studying performance figures ob-

Table 4.7: Speed-up of TREAT and OOPS5 over OPS5

Production System Program	MAB	WALTZ	MAPPER	RUBIK
TREAT (not sorted)	1.48	0.65	1.14	not tested
TREAT (sorted)	1.67	1.48	1.14	not tested
TREAT's Speed-up Factor	1.12	2.28	0.94	not tested
OOPS5 (not sorted)	0.97	0.56	0.94	0.72
OOPS5 (sorted)	1.03	0.61	0.95	0.85
OOPS5's Speed-up Factor	1.06	1.09	1.01	1.18

sorted and non-sorted join sequence. From this table it can be seen that, in general, there is agreement between TREAT and OOPS5 in concluding that sorting of condition elements of production according to the respective sizes of their AVL trees results in some speed-ups. However, the speed-up factor obtained when WALTZ was executed using TREAT is much higher than the OOPS5. Recall that OOPS5 uses the same approach of TREAT in that it examines the conflict set when a WM element that matched a positive CE-Object is removed. Noting that RUBIK and MAPPER have more complex CE-Objects and many more WM elements matching CE-Objects than than WALTZ, we conjecture that the greater speed-up obtained by TREAT over OOPS5 in the case of WALTZ is attributable to a different initial configuration of working memory.

Sorting of a production's CE-Objects in ascending order of the size of their AVL trees before carrying out join has benefited MAB, WALTZ, and RUBIK much more than MAPPER (see speed-up factors in line 6 of table 4.7). This is attributed to the fact that these systems have more joinable CE-Objects than nonjoinable ones and especially in the case of RUBIK as was discussed in section 4.1.2. Hence, this implies that this approach of sorting helps in narrowing down the number of WM-tuples used in the join as the join process progresses from one CE-Object to the next.

MAPPER which is the largest rule set used and takes the longest time to execute has shown a very similar performance to Rete which is mainly attributed to the power of the AVL queries used on the large size of AVL trees (e.g. 576 WM-tuples) of respective CE-Objects and the direct examination of the conflict set in both cases when a working

tained by Miranker.

memory element that matches a positive or negative CE-Object that is either removed or added.

4.3 Conclusion

In this chapter, static and dynamic performance measurements tools were used to assist in studying characteristics of production systems and their behaviour at run-time and particularly from the perspective of the object-oriented transformation and execution of OPS5 production systems. One very important fact found when implementing the performance measurements tools was the ease of embedding them in OOPS5 which is attributable to the object-oriented design and style of programming.

The static tools were mainly used to deal with issues related to the complexity of the left and right hand sides of productions have been found to be variable from program to program. But, in general, production systems have been found to agree on the following observations:

- The number of positive CE-Objects is higher than negative ones and this was considerably higher in the case of RUBIK.
- The number of joinable CE-Objects is higher than nonjoinable ones and this was also considerably higher in the case of RUBIK.
- The advantages of classification of CE-Objects into joinable and nonjoinable ones has found its fruits in join and direct examination of conflict set.
- The number of working memory actions in the right hand sides of productions that add working memory elements are higher than the ones that delete them after the splitting of modify action into a delete followed by an insert action. RUBIK has displayed the extreme of this observation by having an average of 11.19 addition actions per production compared to 2.96 deletion actions on average.
- The degree of uniqueness of CE-Objects has been found to be high and had an impact on improving the performance of production systems and in the majority

of the systems this impact was found to considerable and on average the speed-up factor was found to be 1.40.

The dynamic measurements tools introduced in this chapter have been found to be very helpful in understanding the behaviour of OPS5 production systems in general and OOPS5 in particular. The following observations are a summary of the behaviour obtained by using these tools.

- The state transition diagram for a particular run of a production system program may be considered as a valuable tool in that it describes briefly and deeply such a behaviour in a particular run irrespective of the underlying algorithm employed by the inference engine. This tool may be very beneficial in real–time systems where production systems constitute a major role in the run-time behaviour of such systems. Moreover, the discovery of the join set has closed the gap between the affect set and the conflict set, led us to concentrate on the characteristics of the productions in the join set to improve performance and has suggested using this set as it is a better indicator of production-level parallelism than the affect set used by Gupta irrespective of the underlying algorithm used by the inference engine.

- The movements into and out of the conflict set tool have led us to find a valuable ratio which may be of great importance to the programmer in that it points out the lost computations and as a remedy it was suggested to make left hand sides of productions more specific. Also, this tool has led us to conclude that for an inference engine to avoid the recomputation of the join between CE-Objects of a production as a result of one of these CE-Objects either positive or negative matched by WM element that has been removed or added, respectively, this inference engine has to employ the direct examination of the conflict set. Hence, OOPS5's novel solution to this problem is a major advance over the approaches in TREAT and Rete, considering the fact that there is a potential number of these incidents found in the four production system program studied in this thesis.

- The majority of the time utilised in executing OOPS5 is involved in join computations and is the majority in the case of MAPPER which is the largest and most complex rule set.

In general, OOPS5 turned to be slower than Rete for production systems which have either small rule-sets (MAB) or artificial problem domains (RUBIK, WALTZ), but offers comparable performance with a large realistic system (MAPPER).

This chapter has laid out the bottlenecks in production systems performance in general, and in OOPS5 in particular, so as to lay the ground for the concurrent object-oriented execution of OPS5 production systems.

Chapter 5

Concurrent Execution of OOPS5

In the previous chapter, we presented the various static and dynamic measurements that are concerned with the object-oriented execution of OPS5 production system programs and observed that production objects which belong to the join set (i.e. very small number) take a long time either to succeed or fail in entering the conflict set and have at least one of the following characteristics: a high number of joinable CE-Objects and high number of tuples stored in the AVL trees of corresponding CE-Objects. In this chapter, the future message passing mechanism is utilised to improve the performance of OOPS5 in particular, and production system programs in general, to harness different levels of parallelism. Section 5.1 presents an overview of the future concurrency abstraction and its creation control strategy in EuLisp. Section 5.2 reports the implementation of the various levels of parallelism using OOPS5. Section 5.3 is an assessment of the major results obtained.

5.1 Overview of Futures Concurrent Abstraction

Futures was first implemented in Multilisp [16]. The aim of the future construct is to allow concurrent evaluation of expressions by relaxing the precedence constraints imposed on them lexically. In EuLisp as well as in Multilisp, the evaluation of (`future` *expression*) returns a data object called a future which is a place holder for the value of expression until its evaluation is completed. Hence, this allows the concurrent execution

of the process evaluating expression as well as the process which called future. However, if any process tries to access the value of a future, then if the process evaluating that future has not finished evaluation yet, the caller process is blocked until future evaluation is completed. Otherwise, the caller process proceeds when the value of the future is completed. In EuLisp, a request for the value of a future object is performed using the construct (`future-value` *future-object*).

Using the future construct outlined above implies creating a new task every time a call to the future construct is encountered. This is referred to as Eager Task Creation (ETC) in the original semantics of futures in Multilisp. The crucial advantage of futures is that it allows dynamic partitioning of tasks but there has to be control over future creation in order not to swamp the system and to reduce the total overhead from the use of futures. Two strategies have been proposed, Lazy Task Creation (LTC) and Load Based Partitioning (LBP) [32].

LTC entails that a call to (`future` *expression*) should start evaluating *expression* in the current process and save some information about its parent process so that its continuation can be moved to another process should any processor becomes idle.

Using LBP, the decision to create tasks is based on the number of tasks being queued in the system's task queue. In EuLisp, the number of active futures is bound to a global variable that can be obtained by calling the function (`active-futures-count`). Consequently, to control the load threshold, one has to compare the number of active futures, F, against a predetermined threshold value T such that $F \leq T$. In the case that $F > T$, then no more futures can be created and execution of tasks requesting futures proceed sequentially. With respect to the concurrent execution of OOPS5 in this research, the load threshold is set to $(P + 1)$, where P is the number of physical processors that are to be used in carrying out the concurrent experiments, using the same sample of production system programs used for obtaining the measurements in chapter 4. The advantage of using such a limitation is two-fold. First, each processor has a task to perform when it finishes with its current task. Second, it reduces the overhead of forking many futures, which are large objects in EuLisp, since this would consume a lot of memory and hence cause the system to do frequent garbage collections

especially in cases which lend themselves to the divide-and-conquer algorithmic model of computation.

5.2 Concurrent Execution of OOPS5

One of the main issues that arise at first glance when considering the concurrent execution of OOPS5 is the concurrent movements into and out of the `conflict-set` attribute of the CR-Manager object. As a solution to this problem, the conflict-set is enclosed in a critical region and a boolean semaphore is used to control access. To implement this, a new attribute, namely `conflict-set-semaphore`, is added to the CR-Manager class.

In EuLisp, the construct: `(make-semaphore)` returns a semaphore object, `(open-semaphore` *sem* `)` corresponds to the `P(`*sem*`)` standard operation which implies that any process that encounters such an operation should wait if the value of *sem* is 0 and continue otherwise, `(close-semaphore` *sem* `)` corresponds to also the standard operation `V(`*sem*`)` which sets the value of *sem* to 1.

The objective here is to allow one process at a time to update the conflict-set. To do that, whenever a call is made to any of the following three methods: `insert-into-conflict-set`, `remove-from-cs-production` or `check-removal-of-production`, the updates on the conflict-set in these methods is controlled by the conflict-set-semaphore.

The following sections discuss the implementation of the following four levels of parallelism: (i) *CE-Object*, (ii) *Join*, (iii) *Combination of CE-Object and Join level parallelism*, (iv) *Combination of Action, CE-Object and Join level parallelism.* The implementation of these levels was carried out using FEEL, the implementation of EuLisp, that is currently running on the Stardent machine. The Stardent employs a shared memory architecture with 4 processors sharing 64 MB of RAM.

5.2.1 CE-Object Level Parallelism

When a WM action is executed, a set of CE-Objects are sent either `match-insert`—denoting addition—or `remove`—denoting deletion— messages by the same WM-Distributor object. If either the computation of constant tests of a WM element being matched by

a CE-Object is successful or a WM element to be removed is found in the **WM-AVL-tree** of a CE-Object, then the following tasks are to be performed by such a CE-Object.

1. Updating of the AVL trees
2. Updating of match-knowledge of the corresponding productions
3. If necessary, compute the join between this CE-Object and the rest of the CE-Objects of the production[1].

CE-Object parallelism is then defined as the concurrent processing of the computations performed by each CE-Object of the same WM-Distributor (i.e. same entity) when it is asked to process addition or deletion of a WM element. In this way, CE-Object parallelism serves two goals: concurrent updates of the balanced trees of each CE-Object of the same WM-Distributor and concurrent computations of the join that may be performed in consequence, per WM change.

But, what is the implication of CE-Object level parallelism on the correctness of the execution? To study that, one has to consider the following four cases:

1. What happens when a WM element is added that matches a positive CE-Object?
2. What happens when a WM element is removed and is already matched by a positive CE-Object?
3. What happens when a WM element is added that matches a negative CE-Object?
4. What happens when a WM element is removed but has been matched already by a negative CE-Object?

In the first case, a strategy to implement CE-Object level parallelism has to avoid two situations which may lead to incorrect instantiations entering the conflict set:

- Redundant production instantiations entering the conflict-set. Consider, for example, if C_i and C_j are positive CE-Objects of the same WM-Distributor and of

[1]Recall that testing for eligibility of join and then initiating it is done by **test-join-positive** for positive productions and **test-join-mixed** for negative or mixed productions.

the same production, P_x, which have updated their AVL trees with the WM-tuple, T_s. If C_i computes join with C_j, then the join-list of T_s may be consistent with the join-list of T_s in C_j and hence a P_x instantiation enters the conflict-set. Likewise, when C_j computes join with C_i, an additional but incorrect P_x instantiation enters the conflict-set.

- The possibility of an incorrect join computation if C_i is computing join while $C_j's$ AVL trees are being updated.

In the second case, OOPS5 does not perform any join computations, but instead it searches the conflict-set for productions instantiations having the timestamp of the WM element to be deleted and updates AVL trees of corresponding CE-Objects that are already matched by that element. Hence, parallelism of CE-Objects entails that no process can update the AVL trees of a positive CE-Object if another process is reading it and vice versa, to prevent any system errors or an incorrect join. Clearly, the concurrent update of the AVL trees must be prevented and this is simply solved by enclosing the updating of the AVL trees of positive CE-Objects in a critical region. This has led to the addition of an attribute to the CE-Object, namely, `avl-trees-semaphore`.

In the third case, OOPS5 does not do any join computations but the situation could arise where AVL trees of a negative CE-Object are being updated with a newly added WM element at the same time as this element matches a positive CE-Object and is being used to compute consistent variable bindings with WM-tuples of the negative CE-Object. The side effect of this is the same as in the second case and hence the updates on AVL trees of negative CE-Objects are also enclosed in a critical region as in case 2.

In the fourth case, the same two situations to be avoided in the first case apply. For example, if two negative CE-Objects of the same WM-Distributor and of the same production update their AVL trees and then compute join with a positive CE-Object.

To help in executing cases 1 and 4 concurrently, two new sets of CE-Objects are defined: *parallel* and *mutual*. A CE-Object is said to be *mutual* if there exists at least one CE-Object of the same WM-Distributor belonging to the same production. The

Table 5.1: Average number of CE-Objects per type related to a WM-Distributor

Type of CE-Object	MAB	WALTZ	MAPPER	RUBIK
Positive	6.75	9.8	3.52	31.73
Negative	0	1.6	0.71	2.45
Parallel Positive	4	1.4	1.98	5.82
Parallel Negative	0	1.6	0.66	0.64
Mutual Positive	2.75	8.4	1.55	25.91
Mutual Negative	0	0	0.05	1.82

set of CE-Objects which conform to this criterion is referred to as the *mutual set* and the ones which do not conform are the *parallel set.* Each of these sets is further divided into positive and negative yielding the following four sets: *parallel positive*, *parallel negative*, *mutual positive* and *mutual negative.* On the one hand, this would imply concurrent processing of the insertion of WM elements into the trees of the parallel CE-Objects belonging to the same WM-Distributor or the removal of WM elements from the trees of parallel negative CE-Objects of the same WM-Distributor. On the other hand, the computations related to mutual positive or mutual negative CE-Objects must be processed sequentially to avoid the side effects of cases 1 and 4.

To process cases 2 and 3 concurrently, new sets are identified which contain positive and negative CE-Objects of the same WM-Distributor. These sets are named, *positive* and *negative* CE-Objects. Hence, this would imply concurrent processing of the addition (or removal) of a WM element matching positive (or negative) CE-Objects of the same WM-Distributor.

As a result, this has led to add the following six new attributes to the WM-Distributor class: (1) `parallel positive CE-Objects`, (2) `parallel negative CE-Objects`, (3) `mutual positive CE-Objects`, (4) `mutual negative CE-Objects`, (5) `positive CE-Objects` and (6) `negative CE-Objects`. These sets are all formed at transformation time and there is no overhead at run-time in their creation. Table 5.1 presents the average number of positive, negative, parallel positive, parallel negative, mutual positive and mutual negative CE-Objects per WM-Distributor for MAB, WALTZ, MAPPER and RUBIK.

Table 5.2: Speed-up obtained from CE-Object Level Parallelism over serial OOPS5 and serial OPS5

No. of Processors	MAB	WALTZ	MAPPER	RUBIK
1	0.75 (0.77)	0.95 (0.65)	0.98 (0.93)	0.92 (0.78)
2	0.89 (0.91)	1.07 (0.73)	1.02 (0.93)	1.16 (0.99)
3	0.89 (0.91)	1.09 (0.74)	0.99 (0.94)	1.27 (1.09)
4	0.86 (0.89)	1.13 (0.77)	0.98 (0.94)	1.28 (1.10)

Figure 5-1 presents the algorithm used to implement CE-Object parallelism using these sets where concurrent execution is initiated in the **broadcast-wm-change** method. From this algorithm. it is obvious that the bound on the speed-up obtainable is the time needed to execute the sequential computations in either step 1.c or 2.c. Since the number of negative CE-Objects is much less than positive ones (as can be seen in rows 1 and 2 of table 5.1), it is expected that computations related to mutual positive CE-Objects are the main factor affecting the time taken.

Despite the sequential steps in 1.c and 2.c, it is worth reiterating that the approach taken by OOPS5 of removing production instantiations from the conflict-set in cases 2 and 3 not only benefits the sequential execution of OOPS5—and is also an advantage over OPS5 where it is necessary to recompute join to update the beta-memories—but also its concurrent execution.

Table 5.2 presents the speed-up factors obtained from implementing CE-Object parallelism over serial OOPS5 using the sorted unique CE-Object strategy and over serial OPS5. Thus. a figure less than 1.0 indicates slow-down and greater than 1.0 is speed-up.

RUBIK and WALTZ benefited from CE-Object parallelism whereas MAB and MAPPER did not. The speed-up obtained in these systems is mainly attributable to the concurrent processing of the removal of a WM element matching a large number of positive CE-Objects—especially in RUBIK. MAPPER did not benefit because of the completely different bottleneck it suffers from which was attributed to join computations in section 4.2.3. The performance of MAB degraded because of the small number of WM-tuples stored in the balanced trees and the small number of remove actions.

1. If a WM action specifies deletion of a WM element, then do the following:

   (a) Fork futures fpp_1, fpp_2, $\cdots$, fpp_i, $\cdots$, fpp_n, $1 \leq i \leq n$, to process the removal of this element from AVL trees of the positive CE-Objects, C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, $1 \leq i \leq n$, respectively using the `remove` method.

   (b) Fork futures fpn_1, fpn_2, $\cdots$, fpn_j, $\cdots$, fpn_m, $1 \leq i \leq m$, to process the removal of this element from AVL trees of parallel negative CE-Objects, C_1, C_2, $\cdots$, C_j, $\cdots$, C_m, respectively using the `remove` method.

   (c) Process computations related to the removal of this element from AVL trees of mutual negative CE-Objects, sequentially using `remove` method.

2. Else, If the WM action specifies addition of a WM element, then do the following:

   (a) Fork futures fpp_1, fpp_2, $\cdots$, fpp_i, $\cdots$, fpp_n, $1 \leq i \leq n$, to process computations related to matching this WM element to positive CE-Objects, C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, respectively using the `match-insert` method.

   (b) Fork futures fpn_1, fpn_2, $\cdots$, fpn_j, $\cdots$, fpn_m, $1 \leq i \leq m$, to process computations related to matching this WM element to parallel negative CE-Objects, CE-Objects, C_1, C_2, $\cdots$, C_j, $\cdots$, C_m, respectively using the `match-insert` method.

   (c) Process computations related to matching this WM element to mutual negative CE-Objects, sequentially using `match-insert` method.

In all the above cases, each CE-Object forks futures to update the match-knowledge of corresponding productions (in `productions` attribute of this CE-Object) and computing join related to CE-Objects of these productions if they are eligible for join. Forking of these futures is done before executing an `update-match-knowledge` method and are forced to be evaluated before forcing the evaluation of the forked futures fpp_1, fpp_2, $\cdots$, fpp_i, $\cdots$, fpp_n and fpn_1, fpn_2, $\cdots$, fpn_j, $\cdots$, fpn_m, $1 \leq i \leq m$, $1 \leq i \leq n$.

Figure 5-1: Algorithm employed in implementing CE-Object level Parallelism

Table 5.3: Percentage of Join and Pre-Join computation of the total cost for MAB, WALTZ, MAPPER and RUBIK

Computation	MAB	WALTZ	MAPPER	RUBIK
Pre-Join	75.57%	42.83%	3.9%	42.49%
Join	24.43%	57.17%	96.1%	57.51%

To establish an idea of the maximum speed-up that may be obtained when CE-Object parallelism is exploited, one has to note that this parallelism is not concerned with join computations. Hence, the total speed-up obtained is affected only by the speed-up obtained from the concurrent processing of a single WM change prior to computing join related to that action, if any. To obtain a formula for the speed-up obtained from CE-Object parallelism, assume the following:

C_s is the total cost of execution of OOPS5 sequentially.

$C_s = C_{sp} + C_{sj}$ where C_{sp} is the total cost of computations prior to join computations and C_{sj} is the total cost of join computations sequentially.

C_p is the total cost of execution of OOPS5 using CE-Object level parallelism.

$C_p = C_{pp} + C_{sj}$ where C_{pp} is the total cost of computations prior to join computations using CE-Object parallelism.

Then, the speed-up obtained from CE-Object parallelism over the sequential OOPS5 is:

$$\frac{C_s}{C_p} = \frac{C_{sp} + C_{sj}}{C_{pp} + C_{sj}} = \frac{\frac{C_{sp}}{C_{sj}} + 1}{\frac{C_{pp}}{C_{sj}} + 1} \leq \frac{C_{sp}}{C_{pp}}$$

assuming that $C_{sp} > C_{pp}$. Hence, the speed-up obtained from CE-Object parallelism is bound by $\frac{C_{sp}}{C_{pp}}$.

To have a feeling of the upper bound on the speed-up that may be obtained using CE-Object parallelism in the cases of WALTZ and RUBIK if more processors are employed, the cost of computations prior to join is given in table 5.3[2]. If we ignore the prior to join

[2]The cost of computations prior to join are obtained by summing the percentages in table 4.5 of section 4.2.3 related to the cost of constant tests computations, the updates on AVL trees and the updates on match-knowledge of productions

cost, then the maximum speed-up obtained is 1.75 and 1.74[3] for WALTZ and RUBIK, respectively.

In summary, it may be concluded that the speed-up obtained from CE-Object parallelism is bound by the sequential computations related to mutual CE-Objects and more specifically the cost of join computations. This leads us to investigate parallelisation of these join computations in the next section.

5.2.2 Join Level Parallelism

In section 4.2.3, it was concluded that the majority of the time utilised in executing production systems using OOPS5 is concerned with join computations and, in particular, this was related to positive and negative joinable CE-Objects.

Join related to Positive CE-Objects

In OOPS5, when a WM-tuple w_i matching a CE-Object, C_i, is to be joined with WM-tuples of a positive CE-Object, C_j, of the same production, then the subsequent processing depends on whether C_j is joinable or not. If it is joinable, then an AVL query is run on the AVL tree corresponding to a join variable in the join-list of w_i. If this query is successful, then this results in the set of WM-tuples, w_{j1}, w_{j2}, $\cdots$, w_{js}, $\cdots$, w_{jn}, $1 \leq s \leq n$. At this stage, join level parallelism entails the following:

1. Parallel computation of consistent variable bindings of the join-list of w_i and join-list of $w_{j1}, w_{j2}, .., w_{js}, .., w_{jn}$, $1 \leq s \leq n$.

2. If the consistency check is successful between w_i and any w_{js}, $1 \leq s \leq n$, then a join-list is formed and is used as a seed to compute join with WM-tuples of the next CE-Object in parallel. The subsequent processing is then dependent on the type and joinability of the CE-Object.

On the other hand, if the CE-Object, C_j, is nonjoinable, join level parallelism entails parallel computation of dummy join[4].

[3]These speed-ups were obtained by dividing 1 by the percentage of join cost (e.g. 1 by 0.5717 in WALTZ)

[4]see `join-nonjoin` method in section 3.2.2 between w_i and all WM-tuples of C_j.

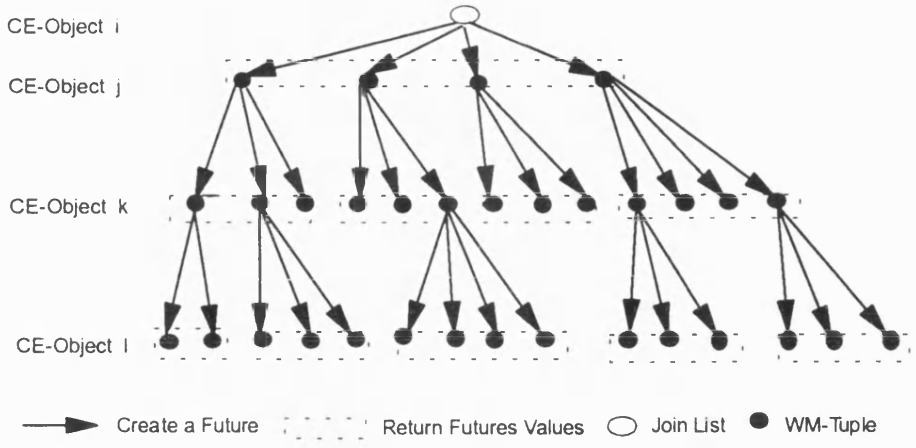

Figure 5-2: Application of Join Level Parallelism with Positive CE-Objects

In the cases, of both joinable and nonjoinable CE-Objects, join-level parallelism fits well with the divide-and-conquer paradigm. Hence, using futures to implement this type of parallelism implies expanding the task tree in figure 5-2 breadth-first by forking join processes until all processors are busy and then expansion proceeds depth-first with the process allocated a single processor. This technique of future creation was referred to as Breadth-first Until Saturation then Depth-first (BUSD)[32]. The way this saturation is controlled is through the use of LBP strategy and hence tasks required by the join level parallelism are dynamically partitioned.

Join Related to Negative CE-Objects

The semantics of negative joinable CE-Objects entail that all WM-tuples w_{j1}, w_{j2}, $\cdots$, w_{js}, $\cdots$, w_{jn}, $1 \leq s \leq n$ that result from running a query on an AVL tree of a CE-Object C_j should fail the variable bindings consistency check in order for the join process not to be terminated[5]. This puts some restrictions on the parallelism that may be obtained, because the join process working on C_j is blocked until one of the following is true:

1. A future returns and signals the success of the variable binding consistency check and in which case the join process is terminated.

[5]see `solve-negative-join` method in section 3.2.2

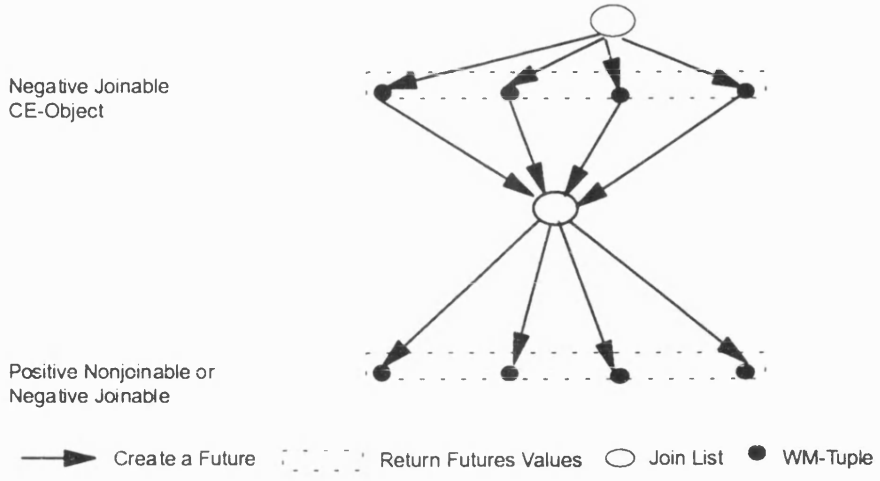

Figure 5-3: Application of Join-Level Parallelism with Negative CE-Objects

2. Or, all forked processes have failed this check and hence the join process continues with the next CE-Object, if any.

In this way, the forked futures which finish early are of no help unlike in the case of positive CE-Objects because the longest executing future represents an upper bound on the parallelism that may be obtained as shown in figure 5-3.

Although intra-node level parallelism[6] is specific to the Rete network and works at finer grain than join-level parallelism, the following two observations are noted when comparing the latter to the former. First, join-level parallelism is more limited than intra-node level parallelism because it is limited to the set of productions sharing a CE-Object[7], whereas the latter is not. Second, Gupta seemed to overlook the limitations inherent in the parallel processing of not-nodes and generalised on two-input and-nodes as if they were the only class of two-input nodes.

Table 5.4 presents the speed-up factors obtained when implementing join-level parallelism using 1 to 4 processors for the four production systems. The following are the remarks concerning this implementation for each of these systems:

[6]Recall from section 2.2.1 that intra-node parallelism implies parallel processing of activations of two-input nodes in the Rete network affected by a single WM change.

[7]These productions are the ones in the productions attribute of the CE-Object

Table 5.4: Speed-up Obtained from Join-Level Parallelism over serial OOPS5 and serial OPS5

No. of Processors	MAB	WALTZ	MAPPER	RUBIK
1	0.90 (0.92)	0.97 (0.67)	0.88 (0.84)	0.97 (0.83)
2	0.89 (0.92)	1.26 (0.87)	1.52 (1.44)	1.04 (0.88)
3	0.86 (0.89)	1.38 (0.96)	1.94 (1.85)	1.06 (0.90)
4	0.83 (0.85)	1.44 (1.00)	2.36 (2.24)	1.04 (0.89)

- MAPPER has the highest speed-up factors, 2.36 and 2.24 over serial OOPS5 and serial OPS5, respectively, using 4 processors. This has resulted in CPU utilisation of 0.59 and 0.56 for OOPS5 and OPS5, respectively. This speed-up is attributed to the concurrent computation of join between WM-tuples of positive CE-Objects. Using more processors is very likely to achieve higher speed-ups given the fact that the average number of WM-tuples used in the join of the bottleneck productions is 326[8].

- The maximum speed-up obtained in the case of WALTZ is 1.44 and 1.00 using 4 processors over OOPS5 and OPS5, respectively. Although a factor of 1.30 is obtained using 2 processors over 1 and a factor of 1.04 is obtained using 4 processors over 3, additional processors are unlikely to achieve a significant speed-up for the following reasons:

  1. In section 4.2.3, it was concluded that the bottleneck in the join process concerning WALTZ was related to the large number of WM-tuples matching negative CE-Objects (i.e. 32 on average). In consequence, the speed-up that may be obtained is bound by either the longest future in the case of failure of computation of consistent variable binding or the earliest future resulting in the success of this computation. In support for this argument, it was found that 90% of the join computations resulted in failure of this computation and hence it was required to wait till the longest future finishes this computation.

[8] see section 4.2.3

2. Join computations constituted approximately 57.15% of the total cost and hence it entails combining this level of parallelism with the CE-Object parallelism to improve on the performance of join level parallelism in the case of WALTZ as will be discussed in the next section.

- In the case of RUBIK, the maximum speed-up obtained using 4 processors is 1.04 over OOPS5 and 0.89 (i.e. slow down) over OPS5. Using more processors is not expected to achieve speed-up for the following reasons:

  1. The average number of WM-tuples used in the join is small (i.e. 10 on average) compared to WALTZ and MAPPER.
  2. As discussed in the case of WALTZ, the cost of join constitutes only 57.31% of the total cost and the rest of this cost is taken in updating AVL trees of CE-Objects and the corresponding match-knowledge of their productions which is shared among large number of WM changes. Hence, this implies combining this level parallelism with CE-Object parallelism or action-level parallelism might improve the performance of such programss as will be discussed in the succeeding sections.

- In the case of MAB, the performance degraded because the number of WM-tuples used in the join process is small, just 1 or 2, and the nature of computation of consistent variable bindings is cheap. In this case, it is obvious that the overhead of creating futures and scheduling them is higher than executing the join tasks themselves.

To conclude this section: join-level parallelism is expected to benefit production system programs whose CE-Objects are matched by very large number of WM-tuples. This is very much the case if records in the database in a real-time system represent the initial configuration of WM of a typical production system program whose task is to implement what-if models (e.g. financial applications). In addition, systems that are temporally redundant (i.e. small number of WM changes per cycle) are also expected to benefit from this level of parallelism. In the case of joining negative CE-Objects,

the speed-up that may be obtained is dependent on either the longest future processing this join in the case of failure of consistency check or the earliest future returning the success of this computation.

5.2.3 Combined CE-Object and Join Level Parallelism

To combine the speed-up obtained as a result of the concurrent updates on AVL trees of CE-Objects and match-knowledge of production[9] with the speed-up obtained from using join-level parallelism, CE-Object parallelism is combined with join-level. A corollary is that the scope of join-level parallelism is extended from being applied on the level of the set of productions sharing a single mutual CE-Object that has been affected by a single WM change to the sets of productions sharing parallel CE-Objects affected by a single WM-change. The following is a summary of the parallelism obtained from this combination:

1. Concurrent processing of the computations related to removal of a single WM element that is already matched by positive CE-Objects of the same WM-Distributor.

2. Concurrent processing of the computations related to addition of a single WM element that match negative CE-Objects of the same WM-Distributor.

3. Concurrent processing of the computations related to addition of a single WM element that match parallel positive CE-Objects of the same WM-Distributor. If any of these objects computes join with other CE-Objects of the same production, then join-level parallelism is applied.

4. Concurrent processing of the computations related to the removal of a single WM element that is already matched by parallel negative CE-Objects and if this results in computing join between CE-Objects of the same production, then join-level parallelism is applied.

5. Concurrent processing of join related to productions whose mutual positive CE-Object is matched by a single WM element added. Each of these productions

[9] see section 5.2 for details on the situations where these updates are applicable

Table 5.5: Speed-up Obtained from Combined CE-Object and Join Level Parallelism over serial OOPS5 and serial OPS5

No. of Processors	MAB	WALTZ	MAPPER	RUBIK
1	0.76 (0.78)	0.94 (0.64)	0.90 (0.85)	0.92 (0.79)
2	0.89 (0.91)	1.38 (0.95)	1.57 (1.50)	1.23 (1.05)
3	0.90 (0.93)	1.42 (0.97)	2.09 (1.98)	1.37 (1.17)
4	0.88 (0.91)	1.71 (1.18)	2.31 (2.19)	1.39 (1.19)

Table 5.6: Speed-up Obtained from Combined CE-Object and Join Level Parallelism over CE-Object Parallelism with respect to OOPS5

No. of Processors	MAB	WALTZ	MAPPER	RUBIK
1	1.01	0.99	0.91	1
2	1.00	1.29	1.54	1.06
3	1.01	1.30	2.11	1.08
4	1.02	1.51	2.36	1.09

computes this join between its CE-Objects using join-level parallelism.

6. Concurrent processing of join related to productions whose mutual negative CE-Object is already matched by a single WM element. Each of these productions computes this join between its CE-Objects using join-level parallelism.

Table 5.5 presents the speed-up factors obtained when applying the combination of these two level parallelism over OOPS5 using the unique sorted CE-Objects and OPS5. Tables 5.6 and 5.7 present the speed-up factors obtained when applying the combination of CE-Object and join levels over CE-Object and join level, respectively. The following

Table 5.7: Speed-up Obtained from Combined CE-Object and Join Level Parallelism over Join Level Parallelism with respect to OOPS5

No. of Processors	MAB	WALTZ	MAPPER	RUBIK
1	0.84	0.97	1.02	0.95
2	1.00	1.10	1.03	1.18
3	1.05	1.03	1.08	1.29
4	1.06	1.19	0.98	1.34

are the main remarks deduced from these tables:

- It is clear that the combination of CE-Object and join level parallelism performed better over all. Hence, it may be concluded that this combination can be expected to benefit production system programs that have one or more of the following characteristics: (1) large number of CE-Objects per production (e.g. RUBIK), (2) CE-Objects are matched by very large number of WM elements (e.g. MAPPER) (3) are temporally or non-temporally redundant (i.e. all programs in the sample).

- Systems that benefited from CE-Object parallelism alone and also benefited from applying join level parallelism alone have benefited from applying both of these two levels together. RUBIK and WALTZ are good examples of this case.

- Using these two levels resulted in obtaining a speed-up factor of between 1.19 and 2.19 over OPS5, ignoring MAB, and a factor of between 1.39 and 2.31 over OOPS5 using 4 processors.

- The loss in performance in MAPPER when using the combination of CE-Object and join level parallelism over join-level parallelism is mainly because MAPPER did not benefit from CE-Object parallelism alone.

- Although, simulations of intra-node parallelism done by Gupta is specific to the Rete algorithm, the speed-up he obtained— a factor of 3.9 on average using 32 processors for the six production systems he studied—is in line with the speed-up obtained from the combination of the two levels studied here if considering the case of MAPPER which requires experimenting with a larger number of processors than used here.

5.2.4 Addition of Action-Level Parallelism

In the previous section, using CE-Object and join level parallelism imposed two constraints. First, sequential processing of computations required by mutual CE-Objects.

Second, the processing of WM actions is done sequentially. This section presents a highly parallel algorithm that avoids these constraints.

In an effort to investigate the parallel processing of activations of two-input nodes in the Rete network, Gupta identified four cases which he recommended not be processed in parallel:

1. Multiple insertion from the left and right sides of a two-input node in the Rete network.

2. Multiple insertions from left input of a two-input node and multiple deletions from the right input of this node.

3. Multiple deletions from left input of a two-input node and insertions from the right input of this node.

4. Multiple deletions from the left and right inputs of a two-input node.

Gupta attributed the inability of processing these cases in parallel to the fact that the Rete algorithm assumes that while a two-input node is being processed, the opposite memory should stay stable. He also noted that there is no simple way to ensure that the opposite memory stay stable and it would be expensive to detect and delete duplicates leaving two-input nodes.

Although the four cases mentioned above are specific to the Rete algorithm, analogous cases have been identified in OOPS5 and resulted in the following three categories:

1. Cases which do not require join between two CE-Objects of the same production. These are: (1) deletion from two positive CE-Objects of the same production or (2) addition of WM-tuples to WM-AVL-trees of two negative CE-Objects of the same production or (3) deletion of a WM-tuple matching a positive CE-Object and addition of a WM-tuple matching a negative CE-Object of the same production.

2. Cases of two CE-Objects where one must compute join with the other but not vice versa. These cases are: (1) adding WM-tuples to WM-AVL-tree of positive CE-Object and then computing join with another CE-Object that updates its

WM-AVL-tree but does not compute join, or (2) removing WM-tuples matching negative CE-Objects while AVL trees of other positive CE-Objects of the same production are being updated.

3. Cases which require join between two CE-Objects of the same production simultaneously. These are: (1) addition of WM elements that match positive CE-Objects while WM elements are removed that match negative CE-Objects of the same production or (2) addition of WM elements matching positive CE-Objects while WM elements are added that match another positive CE-Object of the same production.

In the first category, no join computations are performed, but semaphores are used to protect updates on AVL trees of CE-Objects to ensure that one process at a time updates AVL trees of the same CE-Object.

In the first case of the second category, where one CE-Object computes join with another one which is concurrently updating its AVL trees, the resulting (incorrect) join-list may lead to the incorrect insertion of production instantiation into the conflict-set. This is because the associated join-list was a result of a negative CE-Object matched by a newly added WM element or a positive CE-Object was matched by a WM element which has been removed. In the second case of this category, redundant production instantiations may enter the conflict-set as a result of processing of multiple removals of WM elements matching a negative CE-Object. Consider for example, if a production, P_x, has CE-Objects C_i and C_j which are positive nonjoinable and negative nonjoinable, respectively. If w_1 and w_2 are the only WM elements in AVL trees of C_j and are removed in parallel, then the join between C_j and C_i would lead to a number of P_x instantiations equivalent to twice the number of the WM-tuples stored in the WM-AVL-tree of C_i while the correct outcome of this join is a number of P_x instantiations equivalent to the number of WM-tuples in the WM-AVL-tree of C_i. The solution to this case will be addressed later.

To study the implications of the cases in the third category, let us recapitulate how join is initiated in OOPS5. Join happens only as a result of either of the following

two cases: (1) A WM element has been added that matched a positive CE-Object and its corresponding production(s) is (are) eligible to compute join or (2) A WM element that has been removed but matched a negative CE-Object and its corresponding production(s) is (are) eligible for join. The following two cases are formulated to show that if action level parallelism is to be applied, care has to be taken in processing these cases concurrently:

1. One or more newly added WM elements matching a positive CE-Object, C_i, are to be joined with WM-tuples matching a positive CE-Object, C_j while newly added WM elements matching C_j are to be joined with WM-tuples stored in AVL trees of C_i.

2. One or more newly added WM elements matching a positive CE-Object, C_i, are to compute join with WM-tuples matching a negative CE-Object, C_j while one or more WM-tuples are being removed from C_j's AVL trees.

In both cases, concurrent join computations may lead to redundant production instantiations entering the conflict-set. Consider for example, if C_i and C_j are the only CE-Objects of productions, P_x, and if C_i and C_j have updated their AVL trees with two WM elements, then redundant instantiations of P_x enter the conflict-set.

We can eliminate the side effects of the concurrent join computations utilising the following observation:

> *"If WM elements are processed concurrently up to the stage where all CE-Objects matched by these elements have updated their AVL trees and the match-knowledge of their corresponding productions, then concurrent computation of join between (i) positive CE-Objects of the same production in the same cycle should not use those newly added WM-tuples that have already been used in earlier joins with other CE-Objects or (ii) negative CE-Objects should compute the join with those newly removed WM-tuples that have already been used earlier in initiating join with other CE-Objects"*

This tactic can be realized in initiating the join of a production, P_x, by saving the timestamps of WM-tuples incrementally in a list for each cycle. Using this list whenever

P_x decides to initiate join it has to terminate any subsequent join computations related to all WM-tuples of a positive CE-Object identified by the timestamps in that list. On the other hand, P_x should compute join with those elements in that list in the case of negative CE-Objects.

To implement this new approach in OOPS5, a new attribute is added to the productions class to hold this list of timestamps and it is named, `initial-join-timestamps`. A semaphore is added to the production class to allow one process at a time to read and update `initial-join-timestamps` and is named `initial-join-timestamps-semaphore`.

The implementation of the case related to computation of join with negative CE-Objects is more complex than the positive one. This stems from the semantics of negative CE-Objects and is facilitated by the ability to distinguish between joinable and nonjoinable negative CE-Objects. In the case of joinable ones, a new attribute is added to the CE-Object class to hold the list of WM-tuples[10] deleted. This attribute is called `deleted-WM-tuples` and is protected by a boolean semaphore, namely the `deleted-WM-tuples-semaphore` to allow one a process at a time to update this list of deleted tuples. The case of nonjoinable CE-Objects is simpler and requires only checking whether there exist timestamps in the `initial-join-timestamps` or not. If there are any, then this means that the negative CE-Object has WM-tuples stored in its trees and hence the semantics of negative nonjoinable CE-Objects entail terminating subsequent join computations.

To establish an upper bound on the number of productions instantiations that may enter the conflict-set incorrectly, if we did not take account of the above observation, we use the three lemmas in Appendix A to find a bound, utilising the assumption that CE-Objects are positive and nonjoinable. We choose positive nonjoinable CE-Objects because this maximizes the upper bound since this kind of CE-Objects computes a dummy join. The following two theorems are obtained using the three lemmas in appendix A.

Theorem 1 *If a production, P_x, has C_1, C_2, $\cdots$, C_i, $\cdots$, C_n $1 \leq i \leq n$ positive nonjoinable CE-Objects and there are w_1, w_2, $\cdots$, w_i, $\cdots$, w_n WM elements to be added*

[10] Recall that a WM-tuple consists of a timestamp, a free-list and a join-list

that match C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, respectively, then the number of P_x instantiations to enter the conflict-set is the same as the ones that result from processing w_1, w_2, $\cdots$, w_i, $\cdots$, w_n sequentially or in parallel using the incremental save of timestamps strategy.

Proof: From lemmas 1 and 3, it is obvious that the number of production instantiations that enter the conflict-set are the same and are: $(S+1)^n - S^n$
where S is the average number of WM-tuples matching a CE-Object, C_i, before processing w_i.

Theorem 2 *If a production, P_x, has C_1, C_2, $\cdots$, C_i, $\cdots$, C_n $1 \leq i \leq n$ positive nonjoinable CE-Objects and there are w_1, w_2, $\cdots$, w_i, $\cdots$, w_n WM elements to be added that match C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, respectively, then the upper bound on the number of P_x instantiations to be discarded from the conflict-set that result from processing w_1, w_2, $\cdots$, w_i, $\cdots$, w_n in parallel when the incremental save of timestamps strategy is not used is:*

$$n(S+1)^{n-1} - (S+1)^n + S^n$$

where S is the average number of WM-tuples matching a CE-Object, C_i, before processing w_i.
Proof: This formula is directly obtained by subtracting the number of P_x instantiations obtained in lemma 1 from those obtained in lemma 2.

Combining Action, Join and CE-Object level Parallelism

In combining these, the inference engine executes WM changes in two phases: *update* and *join*. In the *update* phase, WM changes are processed concurrently up to testing for eligibility of productions affected by these changes. In the *join* phase the tests are carried out concurrently and then join-level parallelism is used as discussed in section 5.2.2. The last production computing this join signals the start of conflict-resolution computations. The following is a description of the processing that takes place in these phases.

(1) The Update Phase

Fork futures $f_1, f_2, \cdots, f_i, \cdots, f_n$, $1 \leq i \leq n$, to execute the WM changes in the **actions** attribute of a production being operated on in the **fire** method. When a forked future f_i, $1 \leq i \leq n$, executes the **broadcast-wm-change** method of a WM-Distributor, WD_x, then a set of futures $f_{i1}, f_{i2}, \cdots, f_{ij}, \cdots, f_{im}$, $1 \leq j \leq m$, where m is the number of CE-Objects in **CE-Objects-list** attribute of WD_x, are forked to do the following:

1. If the WM change in f_i will add a new WM element, then f_{ij} $1 \leq j \leq m$, $1 \leq i \leq n$, executes the **match-insert** method on the corresponding CE-Object. If the computations of constant tests are successful, then update AVL trees and fork futures $f_{ij1}, f_{ij2}, \cdots, f_{ijk}, \cdots, f_{nmq}$, $1 \leq k \leq q$, where q is the number of productions stored in the productions attribute of the CE-Object being operated on, to update the match-knowledge of these q productions.

2. If the WM change in f_i will remove a WM-tuple, then f_{ij} $1 \leq j \leq m$, $1 \leq i \leq n$, executes the **remove** method on the corresponding CE-Object. If the WM-tuple to be removed exists in the **WM-AVL-tree** of this CE-Object, then update AVL trees and fork futures $f_{ij1}, f_{ij2}, \cdots, f_{ijk}, \cdots, f_{nmq}$, $1 \leq k \leq q$, where q is the number of productions stored in the productions attribute of the CE-Object being operated on, to update the match-knowledge of these q productions.

In this phase, updates on AVL trees of a CE-Object are protected by the **avl-trees-semaphore** to allow one process at a time to read and update these trees. Likewise, the update on the **match-knowledge** of a production is protected using the newly added attribute to the production class, namely, the **match-knowledge-semaphore**.

(2) The Join Phase

In this phase, all forked futures in the update phase are forced to return their values in order to start joining CE-Objects of the same production that has one of its CE-Objects affected by one or more WM changes if that production is eligible to compute this join.

1. Force futures $f_{ij1}, f_{ij2}, \cdots, f_{ijk}, \cdots, f_{nmq}$, $1 \leq k \leq q$, $1 \leq j \leq m$, $1 \leq i \leq n$ to return. A future f_{ijk} returns either of two values: null or information related to

carrying out the join. On the one hand, the null value implies no join computation is needed because one of the following cases has been encountered: either a removal of a WM element matching a positive CE-Object or addition of a new WM element that matched a negative nonjoinable CE-Object. On the other hand, information returned to test a production's eligibility to compute join consists of the following:

(a) name of method to carry out this join (i.e. either **test-join-positive** or **test-join-mixed**).

(b) the timestamp of the processed WM element

(c) name of production

(d) join-list

(e) free-list

The **initial-join-timestamps-semaphore** attribute of a join-eligible production is consulted to obtain access to read and update **initial-join-timestamps** as follows:

(a) Obtain a copy of **initial-join-timestamps** and attach it to the arguments list of either **test-join-positive** or **test-join-mixed** method.

(b) Update **initial-join-timestamps** with the new timestamp already processed so as the next process consulting this list of timestamps should exclude or include the WM-tuples identified by these timestamps depending on whether the subsequent join is positive or negative, respectively.

Now, fork futures f_1, f_2, .., f_l, .., f_p, $1 \leq l \leq p$, where p is the number of productions to tests their eligibility to carry out this join. Throughout the join computations, whether computing dummy join or not, a check is made to ensure that no WM-tuple, w_x, is to be used in the join if its exists in the the copy of **initial-join-timestamps** (argument to these methods). If it exists, then the join process using w_x is terminated.

On the other hand, if a negative join is computed, then the WM-tuples in **deleted-WM-tuples** attribute of the negative CE-Object are to be considered in computing join if the

Table 5.8: Speed-up Obtained from action-level parallelism combined with CE-Object and join Level parallelism over serial OOPS5 and serial OPS5

No. of Processors	MAB	WALTZ	MAPPER	RUBIK
1	0.42 (0.43)	0.61 (0.63)	0.91 (0.86)	0.39 (0.33)
2	0.61 (0.63)	0.89 (0.62)	1.17 (1.11)	0.57 (0.49)
3	0.71 (0.73)	1.19 (0.83)	1.48 (1.41)	0.63 (0.54)
4	0.74 (0.76)	1.72 (1.20)	1.81 (1.72)	0.67 (0.57)

Table 5.9: Speed-up Obtained when using $(n + 1)$ over n processors in Table 5.8

Processors	MAB	WALTZ	MAPPER	RUBIK
2-1	1.45	1.75	1.68	1.46
3-2	1.16	2.33	1.27	1.11
4-3	1.04	1.44	1.23	1.06

following two conditions are true: (1) The WM-tuples in one of the balanced trees of the CE-Object have failed all variable binding consistency checks with join-list (argument to `solve-negative-join` method) (2) timestamps of WM-tuples in `deleted-WM-tuples` exist in the `initial-join-timestamps` (argument to `solve-negative-join` method).

All join computations between CE-Objects of the same production are performed using join level parallelism.

At this stage, it is time to compute conflict resolution and hence the forked futures $f_1, f_2, ..., f_l, ... f_p$, $1 \leq l \leq p$, are forced to return their values.

Table 5.8 presents the speed-up factors obtained over serial OOPS5 and serial OPS5 when implementing this combination of parallelism. It is clear that as the number of processors increases, more speed-up is obtained. The factors in Table 5.9 show this tendency calculating the speed-up factors obtained from using $(n + 1)$ processors over n for n =1, 2, 3. The following remarks have been deduced from these tables:

- Systems that are temporally redundant (i.e. have large number of actions per cycle) are expected to benefit from this algorithm and the speed-up obtained may

rival the speed-up obtained from the combination of CE-Object and join level parallelism. Note in particular the figures for RUBIK.

- The combination of action, CE-Object and join level of parallelism is not suitable for execution on systems with a small number of processors. Hence, it is concluded that the combination of CE-Object and join level parallelism is the best for such a configuration over the uniprocessor version of OOPS5 and OPS5 based inference engines.

- Systems that benefit from join-level parallelism alone are expected to need a fair number of processors to achieve the same speed-up obtained from just join-level parallelism.

Although the combination of action, CE-Object and join level of parallelism displays a high potential degree of parallelism on the surface[11], we conjecture that the speed-up will not be significant over that obtained from combining CE-Object and join level of parallelism for the following reasons:

1. The cost of the update phase is small—and sometimes the cost of join is the dominant factor, for example in MAPPER—hence, the speed-up obtained from concurrent processing of the update phase saturates much faster than the speed-up obtained from the join phase. We beleive that using more processors will benefit the join phase since we do not see saturation so early as in the update phase.

2. The speed-up that can be obtained from the join phase is from computing join initiated by more than one WM change. But, since the size of the join-set[12] per cycle is small (e.g. 2.13 to 4.59 on average for the four programs used here), then the addition of action-level parallelism to the combination of CE-Object and join level parallelism is unlikely to achieve significant speed-up unless the size of the join-set is sufficiently large. WALTZ is an example of a program that had a

[11]This is because this combination concurrently processes matching of WM elements, updates on AVL trees of CE-Objects, updates on match-knowledge of productions and join of CE-Objects of the same production.

[12]Recall that the join-set refers to the number of production objects eligible to compute join.

relatively large join-set (i.e. 4.59 on average per cycle) compared to the others and did show some benefit from action-level parallelism over the combination of CE-Object and join level parallelism.

Although the ideas in this section were developed recently, there has been a fair amount of work done to present these ideas and test their implementation on a small number of powerful processors. It is intended to study the behaviour of the parallelism discussed in this section using more processors as part of future work.

5.3 Conclusion

The concurrent execution of OOPS5 studied and implemented in this thesis has led us to arrive at two conclusions. First, the architecture of OOPS5 is highly suitable for parallel implementation. Second, the degree of speed-up obtained is highly program-dependent mainly because of the intrinsic bottlenecks in the sequential execution of production system programs which has led to the use of parallelism in an attempt to remedy them. The following are the major findings of using this parallelism in OOPS5:

- The application of CE-Object level parallelism resulted in speeding up systems whose productions have a large number of positive condition elements and have a large number of remove actions in its fired productions.
- The application of join-level parallelism benefited systems whose CE-Objects use large number of WM-tuples in join computations.
- The combination of CE-Object and join level parallelism benefited all production system programs studied in this research and was found to be the best in comparison with serial OOPS5 and serial OPS5 using a small number of processors.
- As a further development we attempted to combine action-level and CE-Object and join level parallelism, which should suit production system programs which are non-temporally redundant, have large number of CE-Objects per production and their CE-Objects use large number of WM-tuples in join computations. Production system programs which have a large join-set per cycle are expected to benefit

from this combination of parallelism over the combination of just CE-Object and join level parallelism.

The empirical results of the concurrent execution of OOPS5 seem to be in line with Gupta's simulations of intra-node and action level parallelism (he obtained a maximum speed-up of 10 over serial OPS5) rather than the simulation speed-ups of PESA-1 (1600) and NON-VON (170). The main reason behind the limited but significant speed-up (e.g. MAPPER) obtained by OOPS5 is attributable to the limited parallelism available in OPS5 production system programs and the different intrinsic sequential bottlenecks.

As a result, it would be misleading to generalise on the speed-up obtained by concurrent OOPS5 or any other parallel implementation over serial OPS5 because this speed-up is program dependent. It is more accurate to say that OOPS5 is suitable for parallel implementation and a significant speed-up may be obtained over serial OPS5 for systems that use large number of WM-tuples in join computations. Such systems may be viewed as more realistic applications of expert systems (e.g. expert database systems) which use large search space of short-term knowledge. Hence, it may be concluded that concurrent OOPS5 utilising action, CE-Object and join level parallelism is a real rather than a simulated contribution to the parallel execution of OPS5 production system programs.

Chapter 6

Conclusion

This research reports the experience of taking a well-known and quite a complex problem—the OPS5 inference engine—and reconstructing it with objects in significant depth by means of Booch's object-oriented development methodology. This has resulted in the construction of a software architecture that is extensible and not constrained by special-purpose hardware as in the cases of DADO and PESA-1. The following are the main outcomes of the use of such technology:

- A new object-oriented inference engine, OOPS5, has been synthesized to execute OPS5 production system programs.
- A new algorithm has been developed to execute such programs. This algorithm utilises an important principle in that it avoids the recomputation of join between condition elements of the same production in the following situations: (1) a WM element is to be removed that was matched by a positive condition element (2) a WM element element has been added and matched a negative condition element. It worth mentioning that while the TREAT algorithm cannot avoid the join re-computations in the second case, Rete cannot avoid it in both cases.
- The construction of a *concurrent object-oriented* platform for executing OPS5 production systems to utilise the available concurrency abstractions. In particular, this thesis exploited the futures abstraction in EuLisp.

- OOPS5 is highly suitable for parallel implementations which was demonstrated by the significant speed-up obtained from exploiting the combination of CE-Object and join levels of parallelism.

- It is possible now to resolve the four cases in Rete which Gupta recommended not be solved in parallel. These cases were contrasted to OOPS5 and a solution was obtained and is summarized below.

- The empirical results of the concurrent execution of OOPS5 seem to be in line with Gupta's simulations of intra-node and action level parallelism rather than the simulation speed-ups of PESA-1 and NON-VON machines. In conclusion, the combination of action, CE-Object and join-level parallelism is a real rather than simulated contribution to the parallel execution of OPS5 production system programs.

The following sections summarize briefly the main results of applying Booch's methodology, the use of object-oriented technology to execute OPS5 production system programs, the performance of OOPS5 sequentially and the concurrent execution of OOPS5.

6.1 The Methodology

The thesis reported the experience of applying Booch's methodology for object-oriented development to synthesise an object-oriented OPS5 inference engine, namely OOPS5. On the methodology side, the following are the main remarks:

- The methodology was easy to use in practice and matches with the natural way of looking at problems from an object-oriented perspective.

- The methodology assisted in the design and development of *concurrent* object-oriented platform for concurrent execution of production systems only after the introduction of timestamps and embedding the aspects of concurrency into the dynamic behaviour of objects.

6.2 The Use of Object-Oriented Technology

Object-oriented technology has made it easier to develop a better solution to executing OPS5 rule sets because:

- OOPS5 allows creation of new rules and matching their condition elements with current configuration of WM elements at run-time and this will support the construction of reflective systems.

- A distinction is made between joinable and nonjoinable condition elements had implication for the execution of OPS5 rule sets. *First*, the processing of removals of production instantiations as a result of inserting a WM element matched by a negative CE-Object is dependent upon the joinability of the negative CE-Object. On the one hand, if it is joinable, then this entails computing consistent variable bindings between join-list of the newly matched WM element and the variable bindings of the respective production instantiations in the conflict-set. On the other hand, if it is nonjoinable, then this entails just searching the conflict-set for production instantiations of the negative CE-Object and then removing them, if any. *Second*, the joining of joinable CE-Objects first followed by the nonjoinable ones avoids the unnecessary join computations that may result if nonjoinable CE-Objects are joined first.

- The concept of uniqueness of condition elements is promoted at a higher level than in Rete with respect to sharing of tests in the sense that OOPS5 classifies a condition element as an entity and then looks at its tests, type and joinability.

The resulting performance of OOPS5 serially is comparable to that of Rete for large and realistic rule-sets (e.g. MAPPER).

6.3 The Static and Dynamic Measurements

Static and dynamic measurements tools were introduced to study the characteristics and the behaviour of OPS5 production systems, respectively. One of the major findings

of the static measurements is the degree of uniqueness of CE-Objects introduced in this research has been found to be high and has considerable impact on the performance. On average, the speed-up factor obtained over the non-uniqueness was found to be 1.40 for the four production systems used in this research. On the other hand, the dynamic measurements resulted in obtaining the following tools:

- Production system programs displayed different behaviour at run-time but agreed in one respect in that join computations constitute most of the cost of a cycle and in some cases is the dominant component (e.g. MAPPER).

- The state transition diagram is a valuable tool in that it describes briefly and deeply the behaviour of the execution of production system programs irrespective of the underlying algorithm employed by the inference engine. In this diagram, the join-set is a valuable finding in that it is a better indicator of production-level parallelism than the affect-set chosen by Gupta. This is because the join-set represents the real number of productions computing join whereas the affect-set is the number of productions affected by a WM change.

- The ratio of the movements of production instantiations from and to the conflict set per cycle is a good indicator of how much computations lost in that cycle. As a remedy, it was suggested that if this ratio is high, then the left hand sides of respective productions instantiations leaving the conflict-set (though not as a result of conflict-resolution) should be made more specific.

- For an inference engine to avoid the recomputation of the join between the CE-Objects of a production as a result of: (i) removing a working memory element that matches a positive CE-Object or (ii) adding a working memory element that matches a negative CE-Object, such an inference engine has to consider direct removal of production instantiations from the conflict-set. Hence, OOPS5's novel solution to this problem is a major advance over the approaches in TREAT and Rete.

6.4 The Concurrent Execution of OOPS5

In summary, the results of the concurrent execution of OOPS5 have led us to conclude two things. First, OOPS5 is highly suitable for parallel implementations. Second, the amount of speed-up obtained is program dependent. The following is a brief summary of the major findings from exploiting parallelism using OOPS5:

- The combination of CE-Object and join-level parallelism benefited all production system programs studied in this research and was found to be the best in comparison with serial OOPS5 and serial OPS5. The speed-up obtained here does not saturate rapidly as in CE-Object parallelism and is mainly dependent on the WM-tuples used in the join computations.

- The addition of action level parallelism to the combination of CE-Object and join level parallelism benefited production system programs which have large join-set and are non-temporally redundant.

- We have been able to resolve the four cases that Gupta recommended not be processed in parallel. These cases which deal with parallel processing of multiple activations of two-input nodes form both the left and right hand sides. New cases were formulated when contrasting these four cases to OOPS5 and a solution was found based on the following: (i) positive CE-Objects of the same production in the same cycle should not use those newly added WM-tuples that have already been used in earlier joins with other CE-Objects or (ii) negative CE-Objects should compute the join with those newly removed WM-tuples that have already been used earlier in initiating join with other CE-Objects.

- The empirical results of the concurrent execution of OOPS5 seem to be in line with Gupta's simulations of intra-node and action-level parallelism rather than the simulation speed-ups of PESA-1 and NON-VON.

- It would be misleading to generalise on the speed-up obtained by the concurrent OOPS5 or any other parallel implementation over serial OPS5 because this speed-up is program dependent. It is more accurate to say that OOPS5 is suitable

for parallel implementations and a significant speed-up may be obtained from parallelism through exploiting join level parallelism.

6.5 Directions for Future Research

Although many issues regarding the sequential and concurrent object-oriented execution of OPS5 production systems have been addressed in this thesis, several issues still remain. This section discusses them briefly.

On the expert database systems side, OOPS5's capability to create productions and match their condition elements with existing working memory elements of the same WM-Distributor (i.e. entity) at run-time is valuable if OOPS5 is used as an inference engine on systems involving a large database (e.g. medical, or financial, etc.) to execute what-if models which may lead to the deduction of new knowledge from the database.

In section 2.3.6, the parallel firing mechanism was discussed and one of the problems presented was the synchronisation problem. This problem required constructing a data dependency graph for all productions. The two kinds of nodes in this graph are the WM-node and the P-node which if compared with OOPS5, the WM-node corresponds to the WM-Distributor object and the P-node corresponds to the production object. The CE-Objects-list attribute holding CE-Objects of a particular entity is further divided into positive and negative CE-Objects (as was done while discussing CE-Object level parallelism). The importance of this segregation is to establish the synchronization sets for every production. Hence, all the information required by this graph is readily available in OOPS5 statically. The only addition is a new attribute in the production class to hold the synchronisation sets for a production object. Given, the fact that the information needed to form the synchronisation sets is available in OOPS5 and the parallel firing mechanism does not constrain itself to any matching algorithm, OOPS5 offers a fertile ground for implementation of such a mechanism which has only been simulated to date. This demonstrates the extensibility of OOPS5 to capture other models of concurrent execution of production system programs.

Load Based Partitioning (LBP) was used in this research to control futures creation

dynamically when executing OOPS5 concurrently. It would be interesting to see the behaviour of OOPS5 when executed concurrently using Lazy Task Creation (LTC) and compare performance given the fact that LTC will always perform at least as well as LBP.

Another future research issue is the use of the Linda [5] concurrency abstraction to store WM-tuples matching each CE-Object in a tuple space to replace the AVL trees attributes of the CE-Object class. The `out` operation is used to store WM-tuples matching a CE-Object while the `in` operation is used to remove WM-tuples matching a CE-Object. In addition, the `rd` operation is to be used to run queries on the tuple space of a CE-Object being joined with other CE-Objects of the same production. In this way, it is possible to have a large scale concurrent object-oriented system employing the `futures` and `Linda` concurrent abstractions.

An obvious direction for further work is to implement the concurrent execution of OOPS5 on a distributed memory multiprocessor system. Such an implementation will bring light to an interesting issue, namely, the application of the timewarp mechanism [19] as a synchronisation tool. One possible way of using this mechanism is to fire the first production that gets into the conflict-set either at once or after waiting for a set period of time. In the case that more productions arrive later in the conflict-set but within the same cycle and do not compete with the already-fired production, then time has been saved. Otherwise, the side effects of the already-fired production can be undone by using the rollback mechanism of timewarp. It is worth noting that the cost of this rollback is expected to be higher if the Rete algorithm is used instead of OOPS5. This is because of the large number of computations required to remove the effects on the beta memory nodes used as inputs to two-input nodes in the Rete network.

Appendix A

Lemmas in chapter 5

Lemma 1 *If a production, P_x, has C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, $1 \leq i \leq n$, positive nonjoinable CE-Objects and there are w_1, w_2, $\cdots$, w_i, $\cdots$, w_n WM elements to be added that match C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, $1 \leq i \leq n$ respectively, then if each w_i is processed sequentially, then the number of instantiations of P_x entering the conflict-set is:*

$$(S+1)^n - S^n$$

where S is the average number of WM-tuples matching a CE-Object, C_i, before processing w_i.

Proof:

When processing w_1, then the number of P_x instantiations is:

$$1 \times \overbrace{S \times S \times \cdots \times S}^{n-1}$$

$$= S^{n-1}$$

When processing w_2, then the number of P_x instantiations is:

$$1 \times (S+1) \times \overbrace{S \times S \times \cdots \times S}^{n-2}$$

$$= (S+1) \times S^{n-2}$$

When processing w_3, then the number of P_x instantiations is:

$$1 \times (S+1) \times (S+1) \times \overbrace{S \times S \times \cdots \times S}^{n-3}$$

$$= (S+1)^2 \times S^{n-3}$$

$$\vdots$$

When processing w_{n-1}, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-2} \times S$$

$$= (S+1)^{n-2} \times S$$

When processing w_n, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

Hence, the total number of P_x instantiations resulting from processing w_1, w_2, w_3, $\cdots$, w_n sequentially is:

$$S^{n-1} + (S+1)S^{n-2} + (S+1)^2 S^{n-3} + \cdots + (S+1)^{n-2} S + (S+1)^{n-1}$$

$$= S^{n-1}[1 + \frac{(S+1)}{S} + \frac{(S+1)^2}{S^2} + \frac{(S+1)^3}{S^3} + \cdots + \frac{(S+1)^{n-2}}{S^{n-2}} + \frac{(S+1)^{n-1}}{S^{n-1}}]$$

$$= S^{n-1} \sum_{i=0}^{n-1} \frac{(S+1)^i}{S^i}$$

$$= S^{n-1}[\frac{1 - \frac{(S+1)^n}{S^n}}{1 - \frac{S+1}{S}}]$$

$$= \frac{S^{n-1} - (S+1)^n S^{-n} S^{S-1}}{1 - \frac{S+1}{S}}$$

$$= \frac{\frac{SS^{n-1} - (S+1)^n}{S}}{\frac{S-(S+1)}{S}}$$

$$= (S+1)^n - S^n$$

Lemma 2 *If a production, P_x, has C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, $1 \leq i \leq n$, positive nonjoinable CE-Objects and there are w_1, w_2, $\cdots$, w_i, $\cdots$, w_n WM elements to be added that match C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, then if the AVL trees of each C_i are updated in parallel, then processing join computations of w_i and WM-tuples of C_{n-i+1}, C_{n-i+2}, $\cdots$, C_{n-i+j}, $\cdots$, $C_{n-i+n-1}$ where $1 \leq j \leq n-1$, $1 \leq i \leq n$ in parallel results in the following number of instantiations of P_x entering the conflict-set:*

$$n(S+1)^{n-1}$$

where S is the average number of WM-tuples matching a CE-Object, C_i, before processing w_i.

Proof:

When processing w_1, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

When processing w_2, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

When processing w_3, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

$$\vdots$$

When processing w_{n-1}, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

When processing w_n, then the number of P_x instantiations is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

Hence, the total number of P_x instantiations resulting from processing w_1, w_2, w_3, $\cdots$, w_n concurrently is:

$$n(S+1)^{n-1}$$

Lemma 3 *If a production, P_x, has C_1, C_2, $\cdots$, C_i, $\cdots$, C_n $1 \leq i \leq n$ positive nonjoinable CE-Objects and there are w_1, w_2, $\cdots$, w_i, $\cdots$, w_n WM elements to be added that match C_1, C_2, $\cdots$, C_i, $\cdots$, C_n, respectively, then if the AVL trees of each C_i is updated with w_i in parallel, then processing join computations of w_i and WM-tuples of C_{n-i+1}, C_{n-i+2}, $\cdots$, C_{n-i+j}, $\cdots$, $C_{n-i+n-1}$ where $1 \leq j \leq n-1$, $1 \leq i \leq n$ in parallel using the incremental save of timestamps strategy results in the following number of instantiations of P_x entering the conflict-set:*

$$n(S+1)^{n-1}$$

where S is the average number of WM-tuples matching a CE-Object, C_i, before processing w_i.

Proof:

When each C_i CE-Object is updated with w_i for $i = 1, 2, 3, \cdots, n$, then the size of each of C_i is $(S+1)$

Let T denote the set of timestamps of WM-tuples that are used in initiating join incrementally. T is initialized to nil.

Assume that t_1 gets into T and hence the number of P_x instantiations that will enter the conflict-set is:

$$1 \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-1}$$

$$= (S+1)^{n-1}$$

When w_2 initiates join, the set T is checked and t_1 is found in it and if any subsequent join computations find the WM-tuple of t_1, then these computations are terminated. T gets updated with t_2. Hence, t_2 is to be joined with C_1, C_3, $\cdots$, C_n where each use $(S+1)$ WM-tuples in this join except C_1 which uses S WM-tuples because t_1 is discarded. Hence the number of P_x instantiations is:

$$1 \times S \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-2}$$

$$= S(S+1)^{n-2}$$

When w_3 initiates join, the set T is checked and t_1 and t_2 are found in it and hence any subsequent join computations that find WM-tuples of t_1 and t_2, then these computations are terminated. T gets updated with t_3. Hence t_3 is to be joined with C_1, C_2, $\cdots$, C_n where each use $(S+1)$ WM-tuples in this join except for C_1 and C_2 which use S because

t_1 and t_2 are discarded. Hence the number of P_x instantiations is:

$$1 \times S \times S \times \overbrace{(S+1) \times (S+1) \times \cdots \times (S+1)}^{n-3}$$

$$= S^2(S+1)^{n-3}$$

$$\vdots$$

When w_n initiates join, the set T is checked and t_1, t_2, t_3, $\cdots$, t_{n-1} are found in it and if any subsequent join computations that find the WM-tuples of these timestamps, then these computations are terminated. T gets updated with t_n. Hence, t_n is to be joined with C_1, C_2, C_3, $\cdots$, C_{n-1} where each use S WM-tuples since t_1, t_2, t_3, $\cdots$, t_{n-1} are discarded. Hence the number of P_x instantiations is:

$$1 \times \overbrace{S \times S \times \cdots \times S}^{n-1}$$

$$= S^{n-1}$$

Hence, the total number of P_x instantiations resulting from processing w_1, w_2, w_3, $\cdots$, w_n concurrently is:

$$(S+1)^{n-1} + S(S+1)^{n-2} + S^2(S+1)^{n-3} + \cdots + S^{n-1}$$

$$= S^{n-1}[\frac{(S+1)^{n-1}}{S^{n-1}} + \frac{(S+1)^{n-2}}{S^{n-2}} + \cdots + \frac{(S+1)}{S} + 1]$$

$$= S^{n-1} \sum_{i=0}^{n-1} \frac{(S+1)^i}{S^i}$$

$$= (S+1)^n - S^n$$

Bibliography

[1] G. Booch. Object-Oriented Development. *IEEE Transactions On Software Engineering*, SE-12(2):211–221, February 1986.

[2] H. Bretthauer, H. Davis, J. Kopp, and K. Playford. Balancing the EuLisp Metaobject Protocol. In *Proc. of International Workshop on New Models for Software Architecture*, Tokyo, Japan, Nov 1992.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Addison Wesley, Reading, Mass., 1985.

[4] B. Buchanan and E. Feigenbaum. DENDRAL and Meta-DENDRAL: Their Applications Dimensions. *Artificial Intelligence*, 11, 1976.

[5] N. Carriero and Gelernter D. Linda in Context. *Communications of the ACM*, 32(4), 1989.

[6] J. Dahl and K. Nygaard. SIMULA: an Algol based simulation language. *Communications of the ACM*, (9), 1966.

[7] R. Davis and J. King. An Overview of Production Systems. In *Machine Intelligence*. John Wiley, New York, 1976.

[8] C. L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1979.

[9] C. L. Forgy. Notes on Production Systems and ILLIAC-IV. Technical Report CMU-CS-80-130, Carnegie-Mellon University, Pittsburgh, 1980.

[10] C. L. Forgy. OPS5 Users's Guide. Technical Report CMU-CS-81-135, Carnegie-Mellon University. Pittsburgh, 1981.

[11] C. L. Forgy. The OPS83 Report. Technical Report CMU-CS-84-133, Carnegie-Mellon University, Pittsburgh, 1984.

[12] C. L. Forgy and Gupta A. Measurements on Production Systems. Technical Report CMU-CS-83-167, Carnegie-Mellon University, Pittsburgh, 1983.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementations.* Addison-Wesley, Reading, Massachusetts, 1983.

[14] A. Gupta. Implementing OPS5 Production Systems on DADO. In *IEEE International Conference on Parallel Processing*, 1984.

[15] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1987.

[16] M. Halstead. Multilisp: A Language for Concurrent Symbolic Computations. *ACM Transactions on Programming Languages and Systems*, 7(4), October, 1985.

[17] B. K. Hillyer and D. E. Shaw. Execution of OPS5 Production Systems on a Massively Parallel Machine. *Journal of Parallel and Distributed Computing*, 3(2), June 1986.

[18] T. Ishida and S. Stolfo. Towards the Parallel Execution of Rules in Production System Programs. In *Proceedings of the International Conference on Parallel Processing*, 1985.

[19] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems.* 7(33). 1985.

[20] G. Kahn and J. McDermott. The MUD System. In *First Conference on Artificial Intelligence Apllications.* IEEE Computer Society and AAAI, December 1984.

[21] D. E. Knuth. *Sorting and Searching.* Addison-Wesley, 1973.

[22] T. Kowalski. *The VLSI Design Automation Assistant: A Knowledge-Based Expert System.* PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, April 1984.

[23] J. E. Laird, P. S. Rosenbloom, and Newell A. Towards Chunking as a General Learning Mechanism. In *AAAI National Conference on Artificial Intelligence.* AAAI, 1984.

[24] T. F. Lehr. The Implementation of a Production System Machine. Master's thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, 1985.

[25] M. Lerner and J. Cheng. The Manhattan Mapper Expert Production System. Technical report, Departement of Computer Science, Columbia University, May 1983.

[26] S. Marcus, J. McDermott, R. Roche, T. Thompson, T. Wang, and G. Wood. Design Document for VT. Technical report, Carnegie-Mellon University, Pittsburgh, 1984.

[27] D. McCracken. *A Production System Version of the Hearsay-II Speech Understanding System.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1978.

[28] J. McDermott. R1: A Rule-based Configurer of Computer Systems. Technical Report CMU-CS-80-119, Carnegie-Mellon University, Pittsburgh, April 1980.

[29] J. McDermott. Extracting Knowledge from Expert Systems. In *International Joint Conference on Artificial Intelligence*, 1983.

[30] J. McDermott, A. Newell, and J. Moore. The Efficiency of Certain Production System Implementations. In *Pattern-directed Inference Systems.* Academic Press, New York, 1978.

[31] D. P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* PhD thesis, Graduate of School of Arts and Science, University of Columbia, New York, 1987.

[32] K. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Department of Computer Science, Yale University, October, 1991.

[33] K. Oflazer. *Partitioning in Parallel Processing of Production Systems.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1987.

[34] J. Padget, G. Nuyens, and H. Bretthauer. An Overview of EuLisp. *Lisp and Symbolic Computation*, 6(1-2), 1993.

[35] D. Patterson and C. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, September 1982.

[36] J. Quinlan. A Comparative Analysis of Computer Architectures for Production System Machines. Master's thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, 1985.

[37] R. Ramnaryan, G. Zimmermann, and S. Krolikoski. PESA-1: A Parallel Architecture for OPS5 Production Systems. In *Proceedings of the Nineteen Annual Hawaii International Conference on Systems Sciences*, 1986.

[38] P. Rosenbloom, J. Laird, J. McDermott, A. Newell, and E. Orciuch. R1-SOAR: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture. In *IEEE Workshop on Principles of Knowledge Based Systems*, 1984.

[39] M. Rychener. *Production Systems as a Programming Language for Artificial Intelligence.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1976.

[40] A. Sabharwal, S. Iyengar, C. Weisbin, and F. Pin. Asynchronous Production Systems. *Knowledge-Based Systems*, 2(2), June 1989.

[41] E. Shortliffe. *Computer Based Medical Consultations: MYCIN.* Elsevier, New York, 1976.

[42] M. Stefik and D. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 1984.

[43] S. Stolfo. Five Parallel Algorithms for Production Systems Execution on the DADO machine. In *AAAI National Conference on Artificial Intelligence*, 1984.

[44] S. Stolfo and D. Shaw. DADO: A Tree-Structured Machine Architecture for Production Systems. In *AAAI National Conference on Artificial Intelligence*, 1982.

[45] P. H. Winston. Learning Structural Descriptions From Examples. *The Psychology of computer vision*, 1972.

[46] M. Zloof. Query-by-Example : A Data Base Language. *IBM Systems Journal*, 16(4), 1977.